Template-Based SPA Attacks on 32-bit ECDSA Implementations

Marcel Medwed
`marcel.medwed@gmail.com`

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Graz University of Technology

Master Thesis

Supervisor: Dipl.-Ing. Dr.techn. Elisabeth Oswald
Assessor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl Christian Posch

November, 2007

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Masterarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Marcel Medwed

I would like to thank
My family for a carefree life
My supervisor for always asking the right questions

Abstract

Template attacks were so far only used to attack 8-bit implementations of symmetric cryptographic algorithms. In this field excellent results have been achieved, since these attacks are the most powerful form of simple power analysis (SPA) attacks. This is the first work which applies them to asymmetric cryptographic schemes on 32-bit platforms. The scheme we attack in this work is the elliptic curve digital signature algorithm (ECDSA). There are several reasons for this choice. First, ECDSA can be executed on 32-bit platforms at considerable speed, even without cryptographic hardware extensions. Second, ARM7 based processors, like used in this work, are widely deployed in mobile devices like PDAs and mobile phones. We show that we can extract the private key, used in the signing algorithm, with only one given power trace. Furthermore we show that we can attack SPA resistant implementations. With our attack we can even overcome other countermeasures like base point blinding and scalar blinding. The only effective countermeasures are those which prevent DPA attacks, like the base point coordinate randomization.

Keywords: Template attacks, ECDSA

Kurzfassung

Template Attacken wurden bisher ausschließlich benutzt, um 8-Bit Implementierungen von symmetrischen kryptographischen Algorithmen zu attackieren. Dabei wurden ausgezeichnete Resultate erzielt. Dies ist gut nachvollziehbar, da sie die mächtigste Art von SPA Attacken darstellen. Wir präsentieren die erste Arbeit, welche Template Attacken gegen 32-Bit Implementierungen von asymmetrischen kryptographischen Verfahren einsetzt. Der Algorithmus, den wir im Speziellen attackieren, ist der Algorithmus für digitale Signaturen auf elliptischen Kurven (ECDSA). Für diese Wahl gibt es mehrere Gründe. Zum einen sind 32-Bit Prozessoren ausreichend um gute Performance Ergebnisse mit diesem Algorithmus zu erzielen, auch ohne Hardwarebeschleunigung. Weiters sind ARM7 basierende Prozessoren, wie auch der von uns eingesetzte einer ist, in mobilen Geräten wie zum Beispiel PDAs oder Mobiltelefonen weit verbreitet. Wir zeigen, dass wir den geheimen Schlüssel, der innerhalb des Algorithmus verwendet wird, mit nur einer Strommessung extrahieren können. Weiters zeigen wir, dass auch SPA resistente Algorithmen unserer Attacke nicht standhalten. Sogar Gegenmaßnahmen wie die Multiplikation des Basispunktes oder des Skalars mit einer Zufallszahl verhindern die Attacke nicht. Einzig und allein DPA Gegenmaßnahmen, wie die Randomisierung der Koordinaten des Basispunktes, schützen vor unserer Attacke.

Stichwörter: Template Attacken, ECDSA

Contents

List of Figures

List of Tables

Preface

In the 1970's the first digital signature schemes have been proposed. Since then more and more information was exchanged and stored digitally. This includes sensitive data like financially or legally relevant information. In 1994 the American National Institute of Standards and Technology (NIST) published the digital signature standard (FIPS 186). By now digital signatures are widely used and many countries have already established ways to manage various kinds of bureaucratic workflows digitally and online.
Today even small devices like PDAs or mobile phones can be integrated in the signing process. A significant amount of these devices use 32-bit ARM7 based processors. But also smartcards slowly move towards 32-bit processors. Power analysis attacks should be and are considered as a threat to all this kind of devices. These attacks have been introduced nine years ago by Paul Kocher et al. Since then a vast number of algorithms and hardware implementations have been investigated to their side-channel leakage. Also the methods and techniques have improved. Template attacks present the most powerful SPA attack. They can allow to brake crypto systems with only one given trace. This is of special interest for randomized algorithms like the elliptic curve digital signature algorithm. This thesis copes with practical template attacks on this algorithm regarding modern architectures.
The remainder of the work is organized as follows:

Chapter 1 provides an introduction to cryptography and asymmetric cryptography.
Chapter 2 describes the used hardware and measurement setup. Furthermore we explain how and why we chose the various parameters of both setups.
Chapter 3 focuses on the algebraic structure of finite fields and describes the algorithms used to perform computations in such fields.
Chapter 4 examines groups based on elliptic curves as well as the used algorithms. Furthermore various attacks on the ECDLP are described. At the end it introduces the elliptic curve digital signature algorithm.
Chapter 5 presents the most important power analysis techniques in detail.
Chapter 6 contains the core work of this thesis. It explains the implemented attacks and presents the achieved results.
Chapter 7 summarizes our results and concludes the thesis.

Chapter 1

Introduction

This chapter will deal with important cryptographic basics. First we will describe the adversary model. Afterwards we will take a look at hash functions, symmetric and asymmetric cryptographic systems. Asymmetric cryptography is of particular relevance for this thesis. Hence we will focus more an those systems. Next we will talk about the advantages of elliptic curve based systems compared to RSA or discrete logarithm based systems. At the end we discuss previously done work.

1.1 Communication and Adversary Model

The aim of cryptography is to secure the communication between two or more parties. For the very basic communication model in Figure 1.1 this means that Alice and Bob want to securely exchange information. Eve represents the attacker and wants to eavesdrop, read and/or alter the sent information. Alice and Bob are not necessarily persons, but can as well be a web shop and a browser. In this case Eve might want to intercept the customer's credit card number.
In order for a system/channel to be secure it has to meet one or more of the following goals:

1. **Confidentiality:** Messages exchanged between Alice and Bob are readable for them, but not for Eve.

2. **Integrity:** If a message sent from Alice to Bob was altered by Eve, Bob is able to detect it.

3. **Authenticity:** If Bob receives a message from Alice, he can be sure that it was indeed Alice who sent the message.

4. **Non-repudiation:** If Alice sends a message, she cannot later on deny that she was the sender. Furthermore, Bob can also prove to a third party that the message originated from Alice.

In the above model Eve is considered to be a powerful attacker. She can eavesdrop the channel as well as inject messages into the channel. Furthermore she has significant computational power and the knowledge about every algorithm used to secure the communication. The only thing Eve is not able to access are all kinds of secret information, like a secret key used within the algorithms.

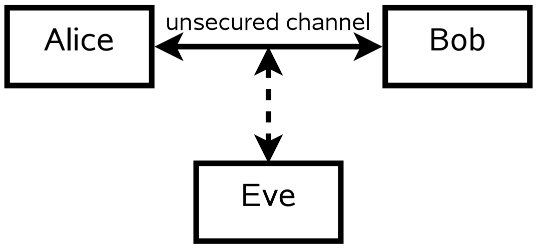


Figure 1.1: Basic communication model

1.2 Hash Functions

Hash functions or also called compression functions take a message of arbitrary bitlength and return a fixed length bitstring $e = H(m)$ as output. Every message has only one hash sum, but since the definition set can be much larger than the image set of the function, two messages may result in the same hash sum. However, although such collisions occur, an attacker must not be able to find them. In order to be of cryptographic interest hash functions have to fulfill certain security properties:

1. **Pre-image resistant:** Given a hash sum h, it is computationally infeasible to find a message m where $H(m) = h$.
2. **2nd pre-image resistant:** Given a message m_1 which results in a hash sum $h_1 = H(m_1)$, it is computationally infeasible to find a message m_2 such that $H(m_1) = H(m_2)$.
3. **Collision resistant:** The attacker can choose m_1 and m_2 and still must not be able to find a message pair for which the hash sums match.

Be aware that the properties are in ascending order regarding the power of the attacker, i.e. collision resistance implies the other two properties.
In cryptography there are many applications for hash functions. For instance they can be used to generate keys from certain input material. Furthermore they can be used as a part of mechanisms that ensure authenticity or integrity. If the hash sum of a message is encrypted, an attacker cannot alter the hash sum and therefore the message without being detected. Even more if the recipient can be sure that only the assumed sender was able to encrypt it, authenticity can be guaranteed.

1.3 Symmetric Cryptography

There are basically two kinds of cryptographic systems: symmetric and asymmetric ones. In symmetric systems Alice and Bob share the same secret key. In asymmetric systems there exists a key pair of which one key is private (i.e. secret) and the other one is public (i.e. accessible to everybody).

Figure 1.2 shows a symmetric setup. First Alice and Bob have to exchange key material over a secret and authenticated channel. Subsequently they can encrypt and decrypt messages with secret-key algorithms and hence achieve confidentiality. Alice therefor uses

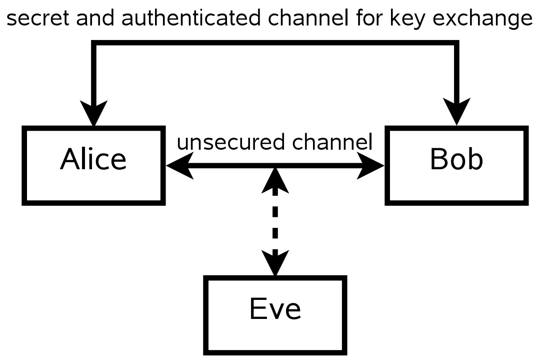


Figure 1.2: Basic symmetric model

an encryption function $E(m,k)$, which takes a message m and the key k and outputs a ciphertext c. Upon receipt of the ciphertext, Bob can use the decryption function $D(c,k)$ to recover the message again.
If authenticity or data integrity are needed Alice and Bob can use the shared secret to generate an authentication tag t. Therefor a message authentication code (MAC) algorithm can be used. It takes a key and the message and outputs $t = MAC_k(m)$. Upon receipt of t and m Bob can calculate $t' = MAC_k(m)$ and check if $t = t'$. Be aware that this does not provide non-repudiation, since it cannot be proven to third party whether the message origins from Alice and not from Bob. Further a third party should never be in possession of the key.
A major drawback of symmetric encryption is the key distribution problem. For military purposes for instance the key is sometimes delivered by a trusted courier. However, for the parties in a vast network like the Internet this is infeasible. Even worse, if N parties want to communicate, $N(N-1)/2$ secret keys are needed. This problem can be addressed by asymmetric crypto systems.
However, the efficiency and the speed of symmetric systems is outstanding compared to asymmetric systems. Also the key sizes are smaller. This is because symmetric systems are designed towards the absence of short-cut attacks. That is, the only way to break such systems is by brute-forcing (trying all possible keys).

1.4 Asymmetric Cryptography

In asymmetric systems each party possesses its own private key d and shares its public key pk with all other parties. A message encrypted with the public key can only be decrypted with the private key of the key pair and the other way around. Figure 1.3 points out that there is no need for a secured channel for the key exchange. This is because it is computationally infeasible to derive the private key solely from the public key. However, it is important that the public key of Alice, Bob uses to encrypt messages, is authentic. Otherwise Eve could make Bob believe that her public key is Alice's. Hence Eve would be able to decrypt confidential messages addressed to Alice.
Public key systems do not only help to achieve confidentiality, but also provide an elegant way to achieve data integrity, authenticity and non-repudiation. If it is ensured that the public key is authentic, this can be done with the help of digital signatures. That is, Alice computes the hash sum of a message m and signs that hash sum with her private key

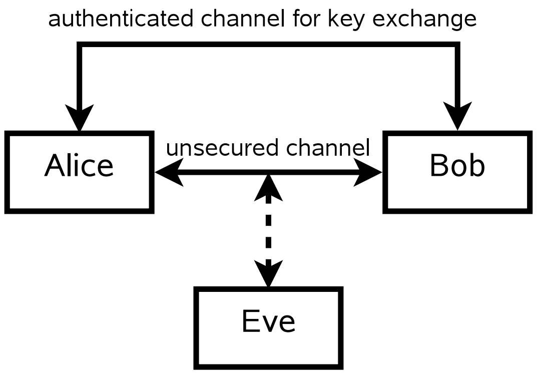


Figure 1.3: Basic asymmetric model

to obtain the signature s. Now everybody in possession of Alice's public key can verify if the signature is valid. The validity of a signature implies that it was indeed message m which was signed and that it was indeed Alice who signed it. This provides integrity and authenticity. Non-repudiation is provided, because any third party can verify the signature as well and since Alice is the only person in possession of the private key, the signature can only originate from her.
The key distribution problem is indeed less complex than for secret-key systems. However, since authenticity is of such great importance, it is not that easy. This problem can be addressed by public-key infrastructures (PKI).
The key lengths of public key systems are in general much longer than for secret key systems. That is because the public and the private key are always related to each other. Although pk can only be determined from d by solving a mathematical hard problem, solving this problem is less complex than brute-forcing. Hence the key size has to be increased until the complexity is comparable to brute-forcing a state of the art symmetric cipher.
Due to the larger key size and the costly algorithms, public key systems are much slower and hence not suited for encrypting large amounts of data. However, since they have significant advantages, hybrid systems are often deployed to combine the advantages of both approaches.

Asymmetric systems can be further divided into their underlying hard problems (Figure 1.4). The first one to have been deployed in an algorithm, namely the Diffie-Hellman key agreement protocol, was the discrete logarithm problem (DLP). It states, that it is computationally infeasible to find x for $y = g^x \mod p$, where p, y and g are given. Of course only if the parameters g and p are appropriately chosen. This was in 1975. Only two years later Rivest, Shamir and Adleman proposed the RSA algorithm for encryption and digital signatures. The RSA problem states, that it is computationally infeasible to find m for $c = m^e \mod n$ if n is the product of two large primes. Again all parameters except m and the factors of n are known. Later on also the discrete logarithm problem has been used for various encryption and digital signature schemes, for instance ElGamal or the digital signature algorithm (DSA).
About ten years later Koblitz and Miller proposed elliptic curves (EC) for public-key cryptography. The deployed problem is called the elliptic curve discrete logarithm problem

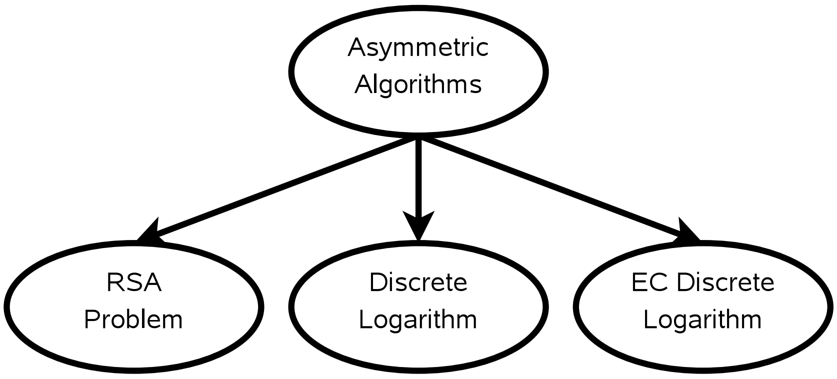


Figure 1.4: Different underlying problems of public-key systems

(ECDLP) and is quite similar to the DLP. Given Q and P for $Q = kP$ it is computationally infeasible to find k (for more details see Chapter 4). Furthermore every DLP based system can be modified in order to be ECDLP based.

1.5 Advantages of elliptic curves

The basic advantage of elliptic curves is the lack of a subexponential algorithm to solve the ECDLP. Hence smaller keys can be used. Table 1.1 compares the needed parameter sizes for different algorithms to achieve an equivalent level of security. If smaller keys can be used, also the required resources are smaller. One such resource is the memory. Another one is the number of needed instructions which relates to the consumed energy. For instance a 192 bit EC point multiplication needs about 200.000 32-bit multiplications, whereas a 1024 bit RSA exponentiation needs about 1.500.000. Although the exponentiation can be sped up, the required memory resources stay the same. Algorithms with small resource requirements are of particular interest for resource constrained systems like mobile devices.

Table 1.1: Needed parameter sizes in bit for different algorithms to achieve equivalent security.

AES		128	192	256
EC size	160	256	384	512
DL group size	160	256	384	512
DL modulus	1024	3072	8192	15360
RSA modulus	1024	3072	8192	15360

1.6 Previous Work

This thesis focuses on power analysis attacks on the ECDLP with the help of templates. As far as we know template attacks have previously only been deployed against 8 bit platforms, but not against 32-bit platforms like our used ARM based processor is. Furthermore we do not know about any previous attempt to use template attacks against asymmetric schemes like the elliptic curve digital signature algorithm (ECDSA). Hence this is the first work to provide practical results on template attacks on 32-bit implementations of asymmetric schemes.

Chapter 2

Setup

This chapter will describe the whole measuring setup. First we will take a closer look at the device under test (DUT). This includes the components of the development board, the LPC2124 microcontroller and its components as well as the used ARM7TDMI-S architecture itself. Afterwards we will focus on the different parts which are involved in the measuring chain, like the shunt, the two probes and the digital oscilloscope. At the end we will characterize the device and determine a suitable power model.

2.1 Device Under Test

2.1.1 Development Board

The device under test is a modified ARM development board manufactured by the company OLIMEX. It features an ARM based microcontroller, providing a JTAG interface, an UART interface and general purpose I/O for communication. Furthermore it contains a wide range of typical microcontroller components like an analog digital converter (ADC) and a watchdog timer. However, these features are not important for our attack.
As a modification, the power supply lines for the core and for the peripheral power supply of the processor have been led out. A picture of the device can be seen in Figure 2.1 and the schematics are shown in Figure 2.8. The figures are further explained by Table 2.1.

Table 2.1: Components shown in the Figures 2.1 and 2.8

1	LPC2124 microcontroller
2	Oscillator crystal
3	Jumper with shunt
4	UART amplifier
5	UART connector
6	JTAG connector
7	Rectifier
8	Voltage regulators
9	Expansion headers
10	JRST jumper
11	BSL jumper

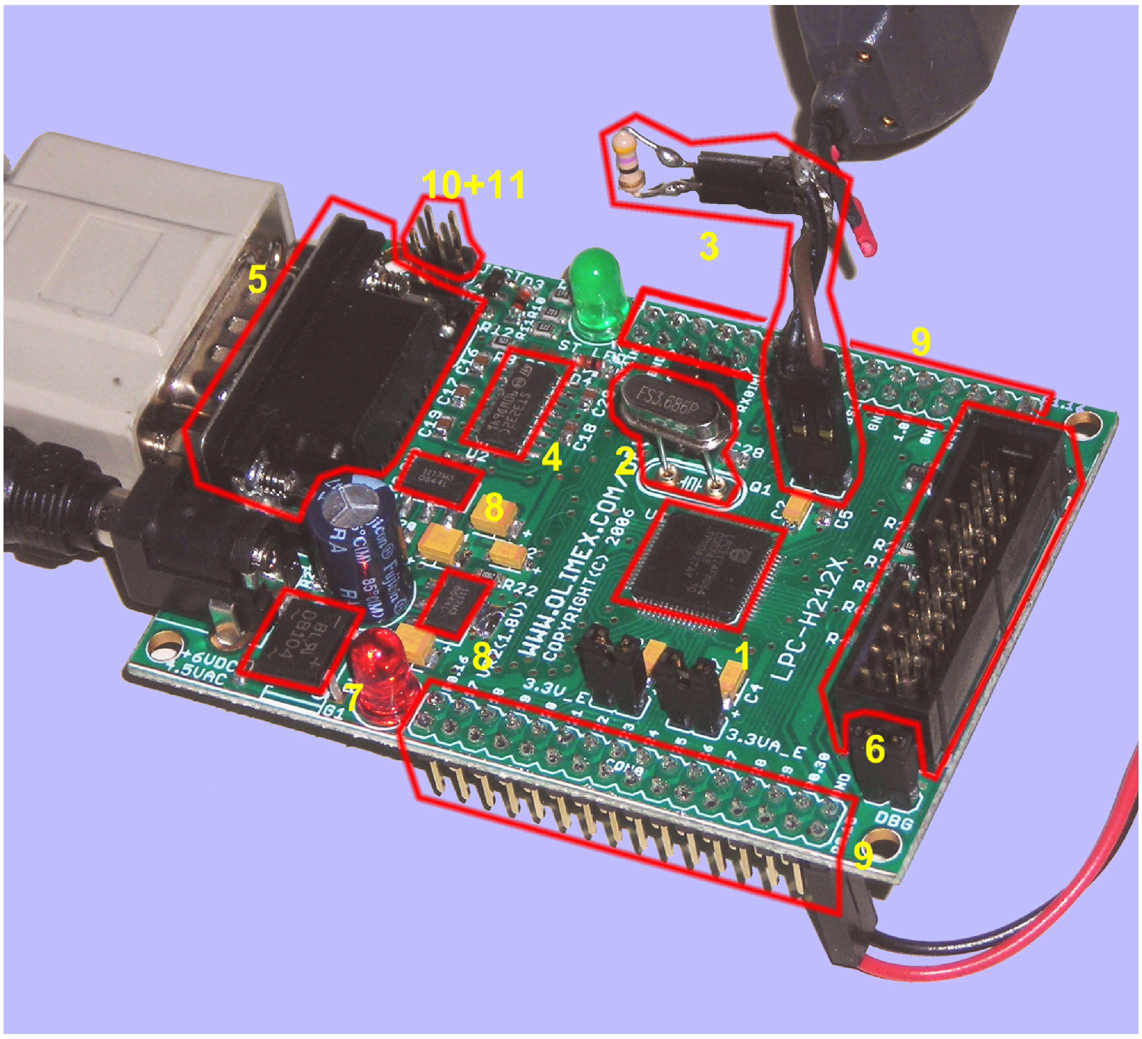


Figure 2.1: The modified ARM development board

2.1.1.1 UART

The serial port of the board is used for two tasks. First, it is possible to flash the processor's internal FLASH memory, to start the execution of program code and to reset the processor via its in-system programming (ISP) interface. Second, the communication between the host PC and the board is done via the serial port.
In order to be able to communicate with the ISP interface the jumpers JRST and BSL have to be closed. The former one hands over the control over the processor's reset pin to the RS232 DTR pin. Whenever a rising edge is sent via the DTR line, a reset is invoked. After a reset, pin P0.14 of the LPC2124 is sampled. If it is low the bootloader enters the ISP mode. Otherwise it tries to find a valid user program. If BSL is closed, pin P0.14 is pulled down, otherwise it is pulled up.
The ISP interface provides several commands to manipulate the RAM and the FLASH and to start the execution at a given address. If the ISP has to be used to start the program, it is crucial that the serial connection is kept open during the control hand over to the host application in order to keep the DTR line high all the time. Otherwise a reset

would be initiated.
A last thing which has to be considered in order to get the serial communication working correctly is the selection of the oscillator crystal. Its frequency has to be dividable by the used baud rate to omit communication errors, see Section 2.1.2.2.

2.1.1.2 JTAG

The JTAG interface represents a very convenient way to program the processor's RAM and FLASH and to directly debug the program on the processor. This is quite useful for short tests, but is not recommended for the actual data acquisition, since the 400MHz USB clock signal is visible in the frequency spectrum of the power consumption signal and hence introduces noise.
Furthermore the startup scripts for the processor can be configured (by defining the macro STARTUP_FROM_RESET) either to immediately start the user program or to loop and stay idle first. In the former case the ability of JTAG debugging is restricted or even impossible, but if no JTAG is used, the startup procedure described in 2.1.1.1 can be omitted. The latter case might be the problem if the processor does not start the user program without JTAG, although everything else is configured and all jumpers are set correctly.

2.1.1.3 Processor

The board features an LPC2124 microcontroller by NXP, which contains an ARM7TDMI-S 16/32-bit core and some peripherals, see Section 2.1.2. Furthermore it has 128 kB FLASH memory and 16 kB SRAM. Both are directly connected to the ARM7 local bus via their controllers. This enables memory transfers without delay cycles.

2.1.1.4 Power Supply

All circuits on the board are supplied by two low-dropout linear regulators of the type LM1117. One is configured for 2.0V and the other for 3.3V output voltage. The former one supplies only the core of the microcontroller. The I/O pads and the other parts of the board like the UART amplifier MAX3232 are supplied with 3.3V. The input voltage range of the regulators is 4.4V to 15V. Considering the rectifier at the input jack of the board and the 1.1V dropout of the regulators, the overall voltage range is about 5.5V to 16.1V.
We used a battery pack as well as a Thandar TS1542S power supply. The choice did not influence the measurement results, since the laboratory power supply already provides a highly stable voltage supply.
Originally the regulator was configured to provide 1.8V output voltage, but it was increased in order to increase the current consumption of the processor and hence increase the signal to noise ratio.

2.1.1.5 Expansion Headers

The expansion headers make most of the package's pins accessible. In our case only pin P0.15 is used to sample the trigger signal. Furthermore a LED is connected to pin P1.25 for some minor visual debugging and status indications.

2.1.1.6 Oscillator Crystal

Due to the layout of the development board, i.e. how the processor is clocked, the applied frequency is restricted from 1 to 30 MHz, see Section 2.1.2.2. In our setup it runs at 3.686 MHz. For power analysis it is crucial to clock the processor with the lowest possible frequency, to prevent the different clock cycles from interfering. If the clock frequency is too high the capacitors in the processor can't be fully charged within one clock cycle. Thus the current drawn is less operation dependent. Figure 2.2 shows two different used clock frequencies and the resulting power consumption plots. The interference can be clearly seen.

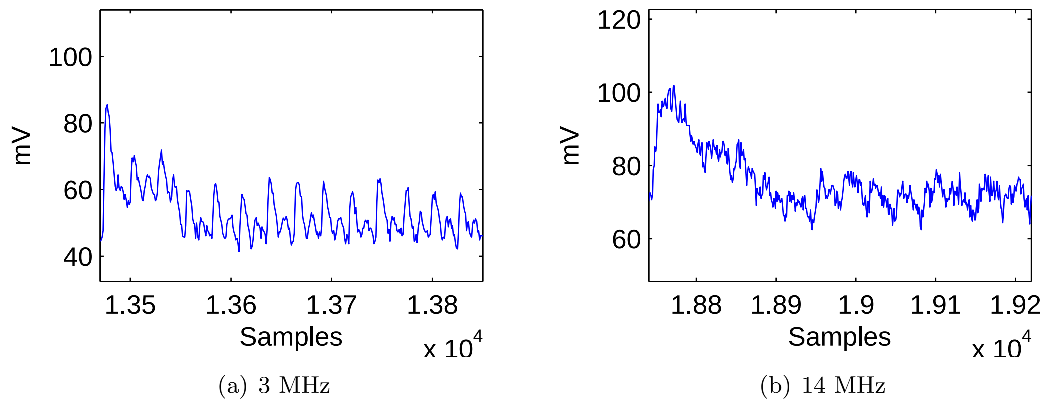


Figure 2.2: Traces for the same instructions at different frequencies

2.1.1.7 Shunt

As a shunt we chose a 40 Ohm resistor for 1.8V supply voltage and 47 Ohm for 2.0V. For an elaboration of the value see Section 2.2.3.

2.1.2 LPC2124 Microcontroller

2.1.2.1 Peripherals

The peripherals of the LPC2124 can be seen in figure 2.3. As mentioned above the ARM7TDMI-S is connected to the FLASH and the SRAM via the ARM7 local bus. The peripherals are connected via an AMBA Advanced High-performance Bus (AHB). Since this microcontroller was designed for low power consumption it allows to turn off unused peripherals via special purpose registers (SPR). The red framed components are those which were switched off in our setup to reduce the noise for the measurements. The external interrupts, the general purpose I/O, the system control, the watchdog timer and the external memory controller cannot be switched off due to the design of the microcontroller. Furthermore the UART cannot be switched off either, because it is needed for the communication.

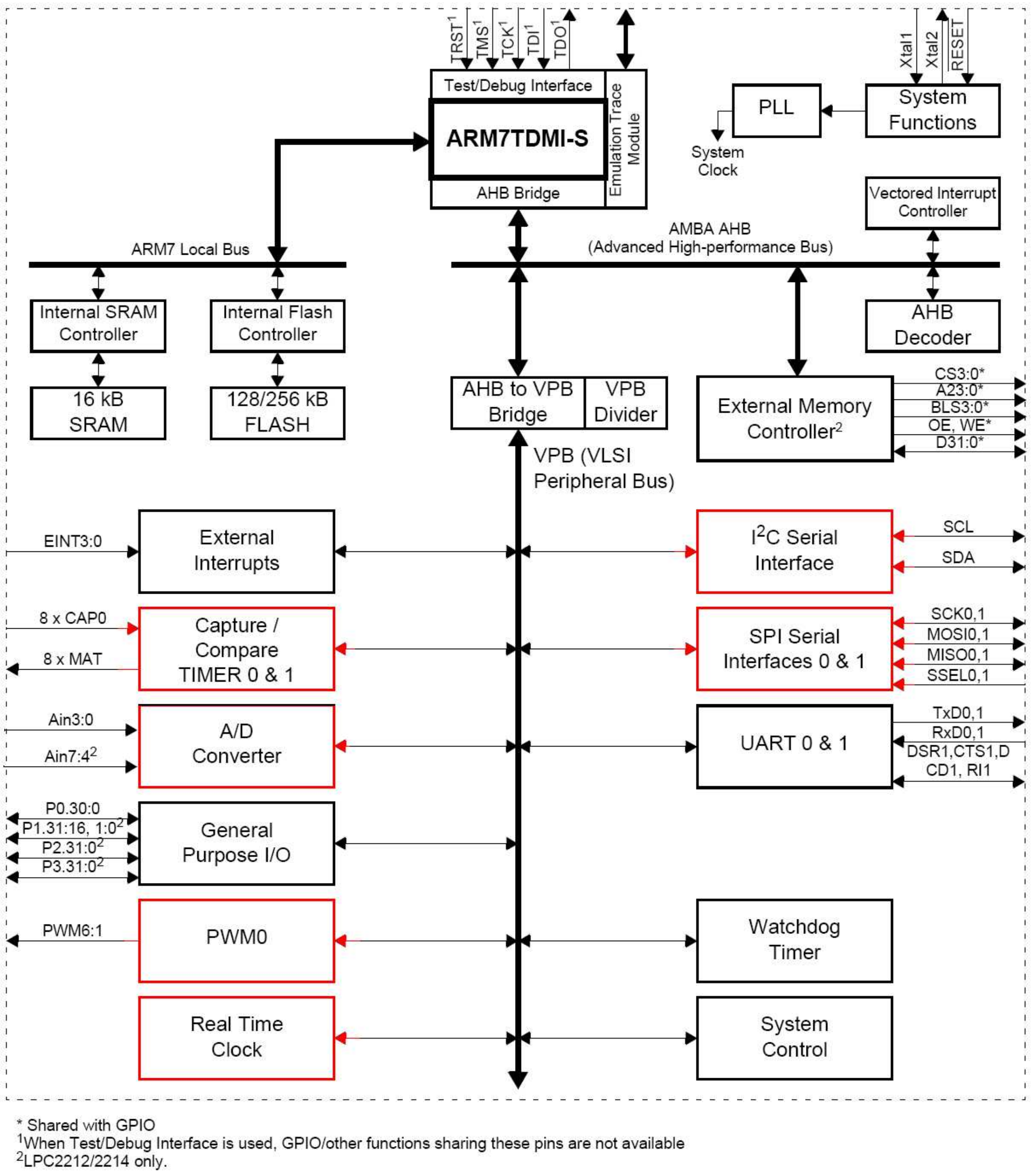


Figure 2.3: LPC2124 block diagram. Taken from [16], p. 19.

2.1.2.2 Clock frequency

The processor takes its clock signal from the pins X1 and X2. It can be operated in slave or in oscillation mode. For the former mode X2 is left unconnected and an externally generated clock signal is applied to X1. For the latter mode an oscillator crystal has to connect the two pins. Basically the clock signal can lie in 3 different frequency ranges. If the processor's internal phase locked loop (PLL) is used or the ISP interface is needed, the applied frequencies must lie between 10 and 25 MHz (according to [16]). If non of those cases applies the processor can be supplied with a frequency between 1 and 30 MHz with an oscillator quartz and from 1 to 50 MHz with a frequency generator. However the figures in the reference manual do not meet reality. The PLL works from 7 MHz upwards and the ISP interface was used even with just 1.686 MHz.

2.1.3 ARM7TDMI-S core

The ARM7TDMI-S is a synthesizable RISC IP core for the use in System-on-Chip (SoC) applications. It follows the ARM v4T architecture and can operate in 16-bit (thumb instruction set) and 32-bit mode (ARM instruction set). Furthermore it has a three stage pipeline and a 32-bit arithmetic logical unit (ALU) including a 32x32 bit multiplier. It is also equipped with a coprocessor interface and a JTAG interface.
The Pipeline consists of a fetch, a decode and an execute stage. Most of the data processing instructions occupy the execute stage only for one cycle. Therefore theoretically one instruction could be finished per cycle.
One of the ALU's input ports has a barrel shifter prepended which allows data processing with implicit shifting. If the shifter operand is given as an immediate value the operation is for free, otherwise it costs one extra cycle.
The multiplier can handle two 32-bit input words and outputs a 64-bit result. The whole multiplication operation needs three to six cycles. It is also able to multiply and accumulate, which takes then four to seven cycles. The execution time varies because the multiplier features an early termination. The multiplication algorithm always processes 32 bit from operand a, but only 8 bit from operand b per cycle. If the most significant byte (MSB) of operand b is zero, the last cycle can be saved, the last two cycles for the two MSBs equal zero and so on.
Load and store instructions occupy the execution stage for more than one cycle. A store needs two (calculate the memory address and write to the memory) and a load needs three (calculate the memory address, read from the memory and transfer the data to the destination register) cycles. The instruction set also has multiple register load and store instructions. They basically only need one more cycle per additional register, but can only be used if the data is sequentially aligned in the memory. These are mostly used for context switches and present the most power expensive instructions. The data processing instructions are the cheapest and the single load and store instructions lie in between.

2.2 Measuring Setup

The measurement setup can be seen in Figure 2.4. The scope is used to send all the control sequences to the board as well as to acquire the data from the board. For the former task it is connected to the board via a serial cable. For the latter one the channels one and two of the scope are used. Channel one is connected to a differential probe, which is indicated by the amplifier symbol in the figure. The differential probe itself is connected to the shunt on the board. The trigger signal is obtained from one of the I/O pins of the processor and is sampled by the passive probe connected to channel two of the scope.

2.2.1 Oscilloscope

The used scope is the DPO 7104 by Tektronix. Our model has a bandwidth of 1 GHz and a 25 megasamples (MS) memory. Basically it is a fast PC with a built-in data acquisition

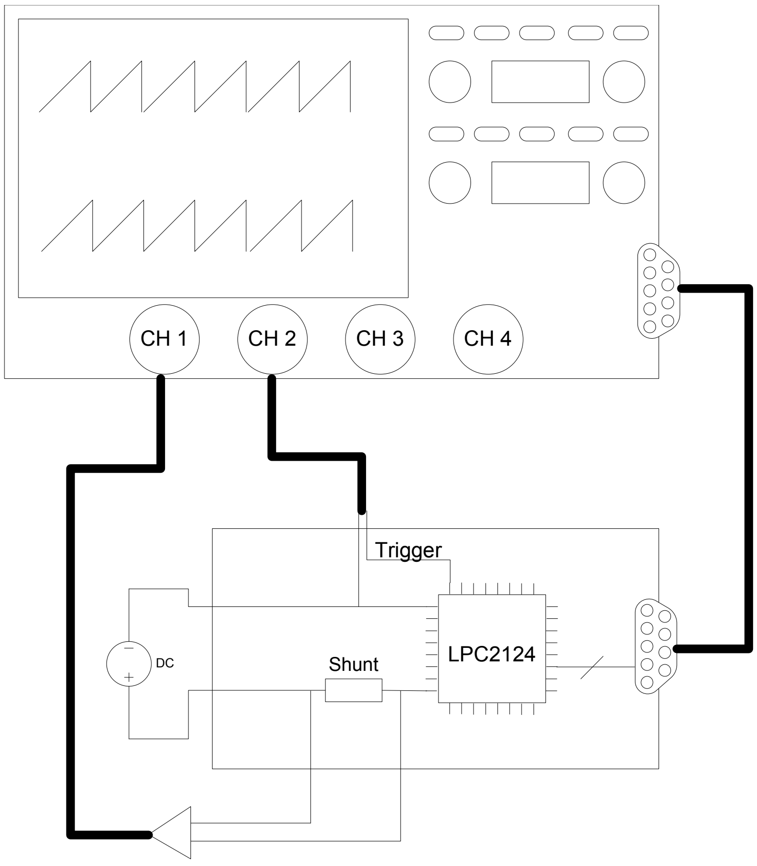


Figure 2.4: The used measurement setup

card. The PC features 2 GB of RAM, a 3.4 GHz Pentium 4 processor and a 60 GB harddisk. Therefore it is well suited for doing all the early data processing steps, like collecting all traces and storing it in special data objects, directly on the scope.
We used the Matlab instrument toolbox to communicate with the TekVisa interface of the scope. This makes it possible to fully control the scope settings, arm the trigger and fetch the acquired data out of Matlab.
The basic setup which was used for the scope (depending on the task some minor changes have been made) can be seen in Table 2.2.

The sampling rate was chosen using heuristics. A rate of 100 Ms/s allows to acquire about 30 points per clock cycle at a clock frequency of 3.686 MHz. Higher sampling rates do not increase the DPA performance significantly. Furthermore the memory is a crucial restriction since execution times of our ECC operations are usually rather long (3.3 million cycles per scalar multiplication). In normal sample mode the available memory allows 737000 clock cycles to be acquired. In the Hi Res mode, the scope averages several points during one acquisition interval. This unfortunately decreases the possible cycles per trace to 184000.
Since the sampling rate is only 100 Ms/s we can limit the input bandwidth of the channel to 250 MHz and therefore reduce the noise. The possible low-pass settings are 20 and 250 MHz, so only the latter one is an option.

Table 2.2: Oscilloscope settings

Sampling rate:	100 MS/s
Acquisition mode:	Hi Res
Trigger type:	Rising edge
Trigger source:	Channel 2
Trigger coupling:	DC
Trigger level:	2.7 V
CH1 Scale:	20 mV/div
CH1 Offset:	28 mV
CH1 Coupling:	DC
CH1 Bandwidth:	250 MHz
CH2 Scale:	5 V/div
CH2 Offset:	0 V
CH2 Coupling:	DC
CH2 Bandwidth:	Full

2.2.2 Probes

2.2.2.1 Differential Probe

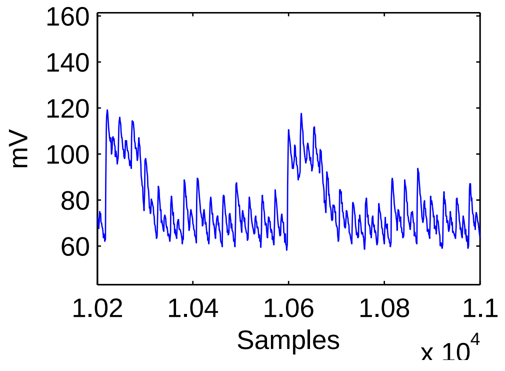
For the data acquisition itself the differential probe P6248 by Tektronix was used. It has a bandwidth of 1.5 GHz, a common mode range of ±7V and a differential mode range of ±850mV. The reason why a differential probe has to be used is that neither of the shunt's ends is connected to a ground potential. Further the ground potentials of all probes are internally connected in the scope and therefore the higher potential at the resistor would be shortcut with the real ground of the board and lead to a damage of the hardware.
The differential probe overcomes this problem by using a differential amplifier, which just amplifies the voltage drop over the shunt and references the result to the ground provided by the scope. Furthermore a property of the differential amplifier is the common mode rejection which allows a higher resolution on the scope.

2.2.2.2 Passive Probe

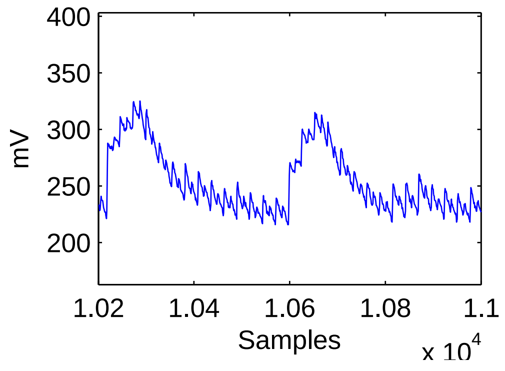
The used passive probe is the P6139A by Tektronix. It has an input bandwidth of 500MHz and a maximum input voltage of 300V. Its task is solely to sample the trigger signal from the CPU's I/O pin.

2.2.3 Shunt

The choice of the shunt is a crucial thing when doing power analysis. The quality of the results heavily depends on this component. But whereas for the quartz we can just go for the slowest possible one there is no such rule for the measurement resistor. However the following things have to be considered: If the resistor is small, the voltage drop is small and the noise level of the differential probe becomes dominant. In other setups known from literature 1 Ohm resistors were sufficient, but our chip does not draw enough current for such a choice. Therefore we rather prefer a large resistor. But the resistor limits the current which is needed to charge the processor's capacitors. Hence if we chose a too large shunt we get a similar effect like when we chose a too fast quartz, i.e. adjacent clock cycles interfere. This can be seen in Figure 2.5, where Figure 2.5(b) looks like a moving average of Figure 2.5(a). The traces were acquired with 1.8V supply voltage, where 47 Ohm were already too much.



(a) 9 Ohm shunt



(b) 47 Ohm shunt

Figure 2.5: Traces for the same instructions with different shunts

In order to select a suitable shunt, several different resistance values have been evaluated. The main criteria was the DPA performance, i.e. the maximal achieved correlation coefficient ρ, of the setup. The results of the measurement series can be seen in Table 2.3.

Table 2.3: Measurement series for different shunts

Resistance [Ω]	Max. Voltage Drop [V]	Max. Current [A]	ρ
3.0	0.054	0.0180	0.26
13.6	0.128	0.0094	0.47
20.4	0.159	0.0078	0.55
27.2	0.192	0.0071	0.58
34.0	0.222	0.0065	0.61
40.8	0.247	0.0061	0.63
47.0	0.274	0.0058	0.60

Table 2.3 points out that a shunt of 47 Ohm at 1.8V limits the current already too much. However if we supply the chip with two Volts we can use the 47 Ohm resistor and get even better results, i.e. a correlation coefficient of 0.72.

2.3 Characterization

This section shows that the Hamming weight model is well suited as a power model for the used microcontroller and determines the signal to noise ratio of the whole setup.

2.3.1 Power Model

To find out if the Hamming weight model is suitable we looked at the power consumption during a memory transfer. Therefore 5000 traces with 50 memory transfers of random data each were acquired. Afterwards a DPA was performed to find the most data dependent points. Those points were then grouped by their Hamming weight and for each group the average was calculated. If the assumption that the power consumption relates to the data's Hamming weight holds, then there should be a linear dependency between these two values.

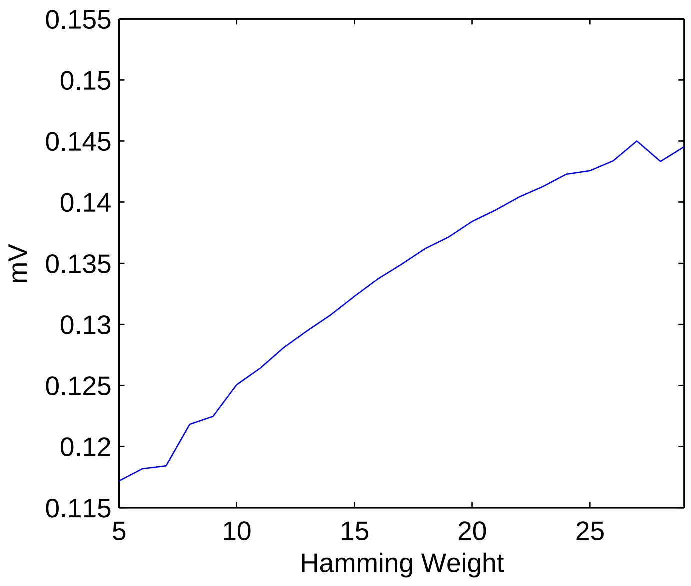


Figure 2.6: Power consumption of the same instruction with different Hamming weights

In Figure 2.6 it can be seen that the assumption holds indeed. The jitters for the large and the small Hamming weights are just caused by noise, because the number of samples for those Hamming weights is much smaller. Therefore the averaging process did not eliminate the noise sufficiently.

2.3.2 Signal-to-Noise Ratio

For the experiment in this section we used 6600 traces with uniformly distributed Hamming weights. In order to calculate the SNR we first need to find suitable points again. Since the points for the highest DPA peak also provide the best SNR we take those again. Then we have to compare the data dependent power consumption variance to the data independent one. Data dependent means that we have to average samples for each Hamming weight in order to get rid of the noise. Afterwards we can calculate a variance that is only data dependent. For the data independent variance we fix the data in order to get the variance of the noise.

$$SNR = \frac{var_{data}}{var_{noise}} = \frac{3.61 \cdot 10^{-5}}{7.29 \cdot 10^{-6}} = 4.95$$

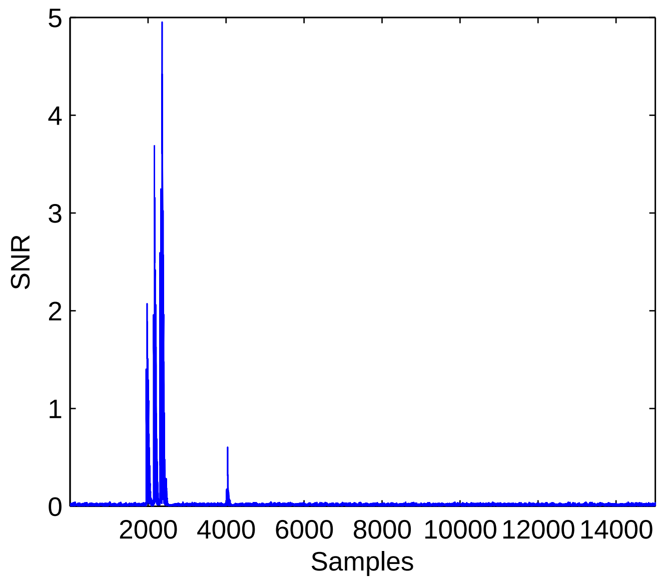


Figure 2.7: The SNR calculated for a whole trace

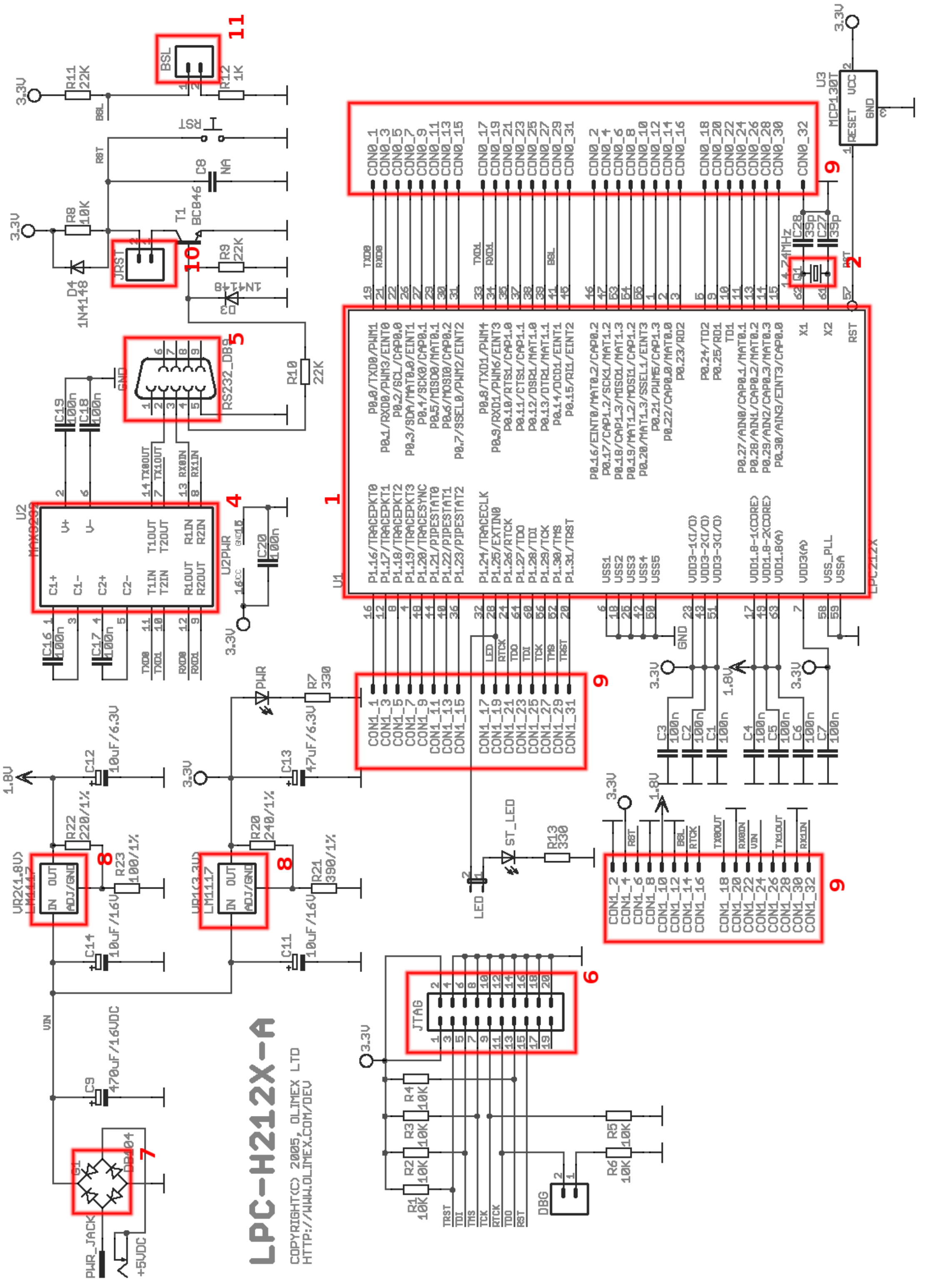


Figure 2.8: Schematic of the modified ARM development board. Based on the schematics provided by Olimex.

Chapter 3

Finite Fields

In this chapter we will first talk about the algebraic structures group, field, and finite field. Afterwards we will take a look at the algorithms that are used to operate on such finite fields and present the basis for the algorithms in Chapter 4.

3.1 Structures

This section presents the definition of an algebraic group, then introduces fields and at the end finite fields, more precisely prime fields. Since the whole thesis only deals with elliptic curves over prime fields, other fields, such as binary extension fields, will be neglected.

3.1.1 Groups

A group is a tuple (G,$\circ$), where G is a finite or infinite set of elements and $\circ$ is a binary operation. Both together have to fulfill the following properties 1-4:

1. **Closure:** A, B $\in$ G $\rightarrow$ A $\circ$ B $\in$ G.
2. **Associativity:** $(A \circ B) \circ C = A \circ (B \circ C)$ $\forall$ A, B, C $\in$ G.
3. **Identity:** $A \circ e = e \circ A = A$ $\forall$ A $\in$ G.
4. **Inverse:** $\exists A^{-1}$ such that $A \circ A^{-1} = A^{-1} \circ A = e$ $\forall$ A $\in$ G.
5. **Commutativity:** $A \circ B = B \circ A$ $\forall$ A, B $\in$ G.

The group operation is typically called either addition or multiplication and furthermore the multiple appliance of this operation is called multiplication or exponentiation. In consequence the group is called an additive group denoted by (G,+) or a multiplicative group denoted by (G,$\cdot$).
If the group also fulfills the fifth property, it is called an abelian group. An example for such a group would be the integers under addition. Under multiplication there would be no inverses, since the inverses would all be rational numbers (except for the number one) and zero does not own one anyway. Hence they can't form a multiplicative group. The same holds for the natural numbers under addition, since the inverses would be the negative numbers, which are not part of the set G then.
Every group also has an order. This is the number of elements in the set G and is denoted by $|G|$. For an infinite set the order is hence also infinite, i.e. $|G| = \infty$.

A finite group is called a cyclic group if there exists a generator g. That means that all elements can be generated by applying the group operation to the generator multiple times. For an additive group this can be written as G $= \{g \cdot n \mid n \in \mathbb{N}\}$ and for a multiplicative group as G $= \{g^n \mid n \in \mathbb{N}\}$.
Furthermore every element of a finite group has an order and this order always divides the group order. It is defined as the smallest number greater zero of times the group operation has to be applied to that element in order to generate the identity element, i.e. $\min_n(\{g \cdot n = e \mid n \in \mathbb{N}\}) \setminus 0$. Hence a generator has the same order as the group. Finally if an element's order is smaller then the group's it generates a subgroup H. Where the order of H of course always divides the order of G, this is stated by Lagrange's theorem.

3.1.2 Fields

A field is a triple ($\mathbb{F}$,+,$\cdot$), where G is a finite or infinite set of elements and both + and $\cdot$ are binary operations. All three together have to fulfill the following properties:

1. **Closure under +:** A, B $\in \mathbb{F} \rightarrow$ A + B $\in \mathbb{F}$.
2. **Closure under $\cdot$:** A, B $\in \mathbb{F} \rightarrow$ A $\cdot$ B $\in \mathbb{F}$.
3. **Associativity for +:** $(A + B) + C = A + (B + C)$ $\forall$ A, B, C $\in \mathbb{F}$.
4. **Associativity for $\cdot$:** $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ $\forall$ A, B, C $\in \mathbb{F}$.
5. **Commutativity for +:** $A + B = B + A$ $\forall$ A, B $\in \mathbb{F}$.
6. **Commutativity for $\cdot$:** $A \cdot B = B \cdot A$ $\forall$ A, B $\in \mathbb{F}$.
7. **Identity for +:** $A + 0 = A$ $\forall$ A $\in \mathbb{F}$.
8. **Identity for $\cdot$:** $A \cdot 1 = A$ $\forall$ A $\in \mathbb{F}$.
9. **Inverse for +:** $\exists A^{-1}$ such that $A + A^{-1} = 0$ $\forall$ A $\in \mathbb{F}$.
10. **Inverse for $\cdot$:** $\exists A^{-1}$ such that $A \cdot A^{-1} = 1$ $\forall$ A $\in \mathbb{F}^* = \mathbb{F} \setminus 0$.
11. **Distributivity:** $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $\forall$ A, B, C $\in \mathbb{F}$.

In other words a field consists of two abelian groups which are defined over the same set and together fulfill the distributive law. As an addition the multiplicative inverse does not need to exist for the additive identity, in fact this is not even possible. To denote the field without the additive identity in the multiplicative group we write $\mathbb{F}^*$.
Furthermore the identity elements have to be distinct from each other, since the additive identity is not a member of the multiplicative group. If it would be, there would have to exist an inverse for it. A little experiment will show that this is not possible. If we want to find n, the multiplicative inverse of the additive identity, we try to add up the additive identity n times in order to obtain the multiplicative identity. Now the identity axiom also states, that the additive identity must not change anything if applied under addition. So there cannot exist an n such that $\sum_n 0 = 1$.
The only way to fulfill the equation above would be to set the multiplicative identity to 0, for a field only containing the element 0 all axioms would hold. For a field containing more elements, this would violate the multiplicative identity axiom ($A \cdot 0 = 0 \neq A$).

3.1.3 Finite Fields

A finite field can be written as $\mathbb{F}_q$ with $q = p^m$, where p is a prime and m an integer. If m equals one, then the field is called a prime field, otherwise an extension field. From above it also follows that the order of such a finite field can only be a prime power. If the power m is one, then it is obvious: Every element in the set has to be co-prime to p in order to have an inverse, plus zero is a member as well. For an $m \geq 2$ the order would be less than q^m unless we represent the field elements in terms of polynomials.
Assume the prime field is $\mathbb{F}_{13}^*$, then the field consists of the integers modulo 13, i.e. $\{0, 1, ..., 12\}$. So the arithmetic operations in that field would be as followed:

- **Addition:** $5 + 16 = 8$ since $21 \mod 13 = 8$.
- **Subtraction:** This can be seen as the addition with the additive inverse of a number. $5 - 16 = 5 + 10 = 2 \mod 13$.
- **Multiplication:** $4 \cdot 5 = 7$ since $20 \mod 13 = 7$.
- **Inversion:** $4^{-1} = 10$ since $4 \cdot 10 = 40 \mod 13 = 1$.
- **Division:** The division is just the multiplication with an inverse.

3.2 Algorithms

The remainder of this chapter will present algorithms for long integer arithmetic in prime fields. Since the numbers we have to deal with are usually much larger than 32 or 64 bit, we have to re-write them as a vector of numbers supported by the processor. For instance, an x-coordinate on an 192-bit elliptic curve can be written as an array consisting of 6 32-bit integers. If m is the number of bits, W the word width and t the number of needed words then $t = \lceil m/W \rceil$. The number is then represented as $\{A_{t-1}, ..., A_1, A_0\}$.

3.2.1 Addition and Subtraction

If we add two 32-bit integers the result can be at most 33 bit large, so the result needs to consist of a carry bit and a 32-bit value. This will be written as $(cout, C) = A + B$. In high level programming languages the carry bit has to be computed explicitly, whereas in assembler most architectures provide a special flag which is set in case of a resulting carry. For subtraction there exists something similar for the case that the subtrahend is larger than the minuend. In such a case the borrow would be set to one. First the basic versions of the algorithms will be presented since both are needed for either of their prime field versions.

Algorithm 1 Long Integer Addition

```
**Require:** Integers $a, b \in [0, 2^{Wt})$
**Ensure:** $(cout, c) = a + b$
  $cout \leftarrow 0$
  **for** $i = 0$ to $t - 1$ **do**
    $c_i \leftarrow a_i + b_i + cout$
    $cout \leftarrow ((c_i < a_i) \vee (cout \wedge (c_i = a_i)))$
  **end for**
  **return** $(cout, c)$
```

Algorithm 2 Long Integer Subtraction

```
Require: Integers $a, b \in [0, 2^{Wt})$
Ensure: $(bout, c) = a - b$
  $bout \leftarrow 0$
  for $i = 0$ to $t - 1$ do
    $c_i \leftarrow a_i - b_i - bout$
    $bout \leftarrow ((a_i < b_i) \vee (cout \wedge (a_i = b_i)))$
  end for
  return $(bout, c)$
```

Algorithm 3 Long Integer Addition in $\mathbb{F}_p$

```
Require: Integers $a, b \in [0, p)$
Ensure: $c = (a + b) \mod p$
  $(cout, c) = a + b$
  if $cout = 1 \vee c \geq p$ then
    $c \leftarrow c - p$
  end if
  return $c$
```

Algorithm 4 Long Integer Subtraction in $\mathbb{F}_p$

```
Require: Integers $a, b \in [0, p)$
Ensure: $c = (a + b) \mod p$
  $(bout, c) = a + b$
  if $bout = 1$ then
    $c \leftarrow c + p$
  end if
  return $c$
```

3.2.2 Multiplication

The field multiplication can be done in the same way as the addition and the subtraction in a field, by computing the result and reducing it afterwards. The only difference is that the result of a multiplication of 2 n-bit words is $2n$-bit long. Hence reducing like above would be way too inefficient. Therefore techniques like in Section 3.2.4 have to be used.

Algorithm 5 Long Integer Multiplication (operand scanning), taken from [5], p. 31

```
Require: Integers $a, b \in [0, 2^{Wt})$
Ensure: $c = a \cdot b$
  $c \leftarrow 0$
  for $i = 0$ to $t - 1$ do
    $u \leftarrow 0$
    for $j = 0$ to $t - 1$ do
      $(uv) \leftarrow c_{i+j} + a_i \cdot b_j + u$
      $C_{i+j} \leftarrow v$
    end for
    $C_{i+t} \leftarrow u$
  end for
  return $c$
```

It can be seen that the algorithm above has a complexity of $\mathrm{O}(n^2)$, where n is the number of bits. This complexity can be reduced with the Karatsuba-Ofman multiplication method. Therefor we have to split our operands in two halves: $a = a_1 \cdot 2^{n/2} + a_0$ and $b = b_1 \cdot 2^{n/2} + b_0$. Then c results in:

$$\begin{aligned} c &= a \cdot b \\ &= (a_1 \cdot 2^{n/2} + a_0)(b_1 \cdot 2^{n/2} + b_0) \\ &= a_1 \cdot b_1 \cdot 2^n + [(x_0 + x_1) \cdot (y_0 + y_1) - x_1 \cdot y_1 - x_0 \cdot y_0] \cdot 2^{n/2} + x_0 \cdot y_0 \qquad (3.1) \end{aligned}$$

The result of Equation 3.1 is obviously the same, but the number of operations is different. Recall that we first had an n-times-n bit multiplication and now have 3 $n/2$-times-$n/2$ bit multiplications. The four additions are cheaper than the multiplication especially for larger operands and the two multiplications by powers of two are just shifts which are cheap as well. This reduces the complexity to $\mathrm{O}(n^{log_2 3})$.

3.2.3 Squaring

Squaring is cheaper than multiplying. Since for instance $a_i \cdot b_j$ is the same as $a_j \cdot b_i$ about half of the multiplications can be saved. On the other hand the overhead caused by the inner loop is significant for small operands. The following algorithm exploits the above mentioned symmetry.

Algorithm 6 Long Integer Squaring, taken from [5], p. 35

```
Require: Integer $a \in [0, 2^{Wt})$
Ensure: $c = a^2$
  $r0 \leftarrow 0, r1 \leftarrow 0, r2 \leftarrow 0$
  for $k = 0$ to $2t - 2$ do
    for each element of $\{(i,j) | i + j = k, 0 \leq i \leq j \leq t - 1\}$ do
      $(uv) \leftarrow a_i \cdot a_j$
      if $i < j$ then
        $(cout, uv) \leftarrow 2 \cdot uv, r2 \leftarrow r2 + cout$
      end if
      $(cout, r0) \leftarrow r0 + v$
      $(cout, r1) \leftarrow r1 + u + cout$
      $r2 \leftarrow r2 + cout$
    end for
    $c_k \leftarrow r0, r0 \leftarrow r1, r1 \leftarrow r2, r2 \leftarrow 0$
  end for
  $c_{2t-1} \leftarrow r0$
  return $c$
```

3.2.4 Reduction

The reduction is a crucial step within the field arithmetic and the overall performance can heavily depend on its efficiency. There are basically three methods to do the reduction in an efficient way. Which one of them is best suited depends on the situation. The fast reduction algorithm is the fastest, but is only possible if the field modulus is a prime of a special form. The Barret reduction would be the best choice if the operands change often or if the operands are not involved in long computations. For exponentiations for example the Montgomery method would be well suited, because the overhead for the required initial transformation into the Montgomery domain pays off.

Fast Reduction: As mentioned above the fast reduction requires a special modulus, so it is not a very general method. However, all the standardized NIST curves have been designed towards the use of this algorithm. These so called NIST primes have the property that they can be written as a sum or difference of powers of two. Furthermore the number of summands is small and their power is almost always a multiple of 32. As an example ([5], p.45) we look at the prime used for the NIST curve P192. Written down in radix 2^{64} it looks as follows:

$$p_{192} = 2^{192} - 2^{64} - 1.$$

The number we want to reduce is at most 384 bit long and can be written as:

$$c = c_5 \cdot 2^{320} + c_4 \cdot 2^{256} + c_3 \cdot 2^{192} + c_2 \cdot 2^{128} + c_1 \cdot 2^{64} + c_0.$$

Thus the congruences for the powers ≥ 192 can be derived as:

$$2^{192} \equiv 2^{64} + 1 \pmod{p}$$

$$2^{256} \equiv 2^{128} + 2^{64} \pmod{p}$$

$$2^{320} \equiv 2^{128} + 2^{64} + 1 \pmod{p}$$

So all that is left to do is to replace the powers of c greater than 128 with their congruences regarding p. The resulting sum is for sure smaller than four times p and can be easily reduced by subtracting p in order to obtain an element of the field.

Algorithm 7 Fast reduction for p_{192}

```
Require: Integer $a = a_{11}, ..., a_0 \in [0, p^2)$
Ensure: $c \mod p$
  $s1 \leftarrow 0, s2 \leftarrow 0, s3 \leftarrow 0$
  $s1_0 \leftarrow a_0, s1_1 \leftarrow a_1, s1_2 \leftarrow a_2$
  $s1_3 \leftarrow a_3, s1_4 \leftarrow a_4, s1_5 \leftarrow a_5$
  $s2_0 \leftarrow a_6, s2_1 \leftarrow a_7, s2_2 \leftarrow a_6$
  $s2_3 \leftarrow a_7, s2_4 \leftarrow 0, s2_5 \leftarrow 0$
  $c = (s1 + s2) \mod p$
  $s1_0 \leftarrow 0, s1_1 \leftarrow 0, s1_2 \leftarrow a_8$
  $s1_3 \leftarrow a_9, s1_4 \leftarrow a_8, s1_5 \leftarrow a_9$
  $s2_0 \leftarrow a_{10}, s2_1 \leftarrow a_{11}, s2_2 \leftarrow a_{10}$
  $s2_3 \leftarrow a_{11}, s2_4 \leftarrow a_{10}, s2_5 \leftarrow a_{11}$
  $s3 = (s1 + c) \mod p$
  $c = (s2 + s3) \mod p$
  return $c$
```

Barrett Reduction: This method works for every modulus and only needs some precomputation for every used modulus, but there is no need to apply any transformation to the operands itself. The idea behind this reduction algorithm is to estimate $q = \lfloor a/p \rfloor$ roughly. Afterwards $p \cdot q$ can be subtracted and the result can then be efficiently adjusted by subtracting p only a few more times.

Algorithm 8 Barrett reduction

```
Require: Integers $a \in [0, 2^{2Wt}), \mu = \lfloor 2^{2Wt}/p \rfloor$
Ensure: $c \mod p$
  $q1 \leftarrow \lfloor c/2^{W(t-1)} \rfloor$
  $q2 \leftarrow q1 \cdot \mu$
  $q3 \leftarrow \lfloor q2/b^{k+1} \rfloor$
  $r1 \leftarrow x \mod 2^{W(t+1)}$
  $r2 \leftarrow (q3 \cdot p) \mod 2^{W(t+1)}$
  $c \leftarrow r1 - r2$
  if $c < 0$ then
    $c \leftarrow c + 2^{W(k+1)}$
  end if
  while $c \geq p$ do
    $c \leftarrow c - p$
  end while
  return $c$
```

q3 is at most the real q plus 2, therefore the result of r1 - r2 is for sure smaller than $2^{W(t+1)}$. The algorithm needs t(t+4)+1 single-precision multiplications.

Montgomery Reduction: This method takes as input a $2\cdot W\cdot t$ bit integer a, a quantity R, the modulus and it's negative inverse modulo R. The return value is $a\cdot R^{-1} \mod p$. R and p have to be co-prime. Therefore if p is odd R can be chosen as 2^{Wt}, this is important for the efficiency of the algorithm as we will see.

Algorithm 9 Montgomery reduction

```
Require: Integers $a \in [0, 2^{2Wt}), R = 2^{Wt}, p' = -p^{-1} \mod R, p$
Ensure: $a \cdot R^{-1} \mod p$
1: $c \leftarrow (a + (a \cdot p' \mod R)p)$
2: $d \leftarrow a/R$
3: if $d \geq p$ then
4:     $a \leftarrow a - p$
5: end if
6: return $d$
```

In line 1 we don't change the value of a, since we just add a multiple of p. This quantity is chosen such that it is $-a \mod R$, which is the last t words of $-a$. The addition therefore ensures that c is dividable by R. Since we chose R as 2^{Wt} this can be efficiently implemented as a right shift. More formally this can be written as:

$$\begin{aligned}
c &= a + (a\cdot p' \bmod R)p \\
&= a + (a\cdot p' + k\cdot R)p \\
&= a + a\cdot p'\cdot p + k\cdot R\cdot p \\
&= a + a\cdot(R\cdot R^{-1} - 1) + k\cdot R\cdot p \\
&= a + a\cdot R\cdot R^{-1} - a + k\cdot R\cdot p \\
&\equiv a\cdot R\cdot R^{-1} \bmod p
\end{aligned}$$

The substitution of $p'\cdot p$ with $(R\cdot R^{-1} - 1)$ is valid, because the fact that R and p are co-prime and p' the negative inverse of $p \mod R$ ensures that $R\cdot R^{-1} - p\cdot p' = 1$.
The only problem left is that we get $a/R \mod p$ instead of a. Therefore we first have to transform the operands for the multiplication into the Montgomery domain, i.e. $\tilde{x} = x\cdot R$. Hence $\tilde{x}^2 = x^2\cdot R^2$. After the reduction step we obtain $x^2\cdot R^2/R$ which is the Montgomery transformed of x^2. So if we use Montgomery transformed operands we always stay in this domain. To obtain such a transformed from a value we just have to multiply it by R^2 mod p and reduce it, to transform a value back we just reduce it once again. So if we define Mont(A,B) as the Montgomery multiplication which already contains the reduction we get:

Table 3.1: Montgomery multiplication behavior

A	B	Mont(A,B)
$\tilde{a}$	$\tilde{b}$	$\widetilde{a\cdot b}$
a	R^2	$\tilde{a}$
$\tilde{a}$	1	a

In order to calculate Mont(x,R^2) we need to precompute R^2 mod p and therefore need another reduction algorithm like the Barrett reduction first.

The number of needed single precision multiplications is with t(t+1) smaller than the one for the Barrett reduction. But we have to consider that this advantage only pays off, if the number of needed transformations is small compared to the Montgomery multiplications.

3.2.5 Inversion

In order to be able to perform divisions in fields we need the inverse of elements. Usually the inversion is the most costly operation in finite fields. The extended Euclidean algorithm computes the greatest common divisor for two numbers x and y and also provides two numbers a and b such that $a \cdot x + b \cdot y = \gcd(x, y)$. For prime fields we set y to p and hence the gcd is 1. Taking this equation modulo p we get:

$$a \cdot x + b \cdot p = 1 \equiv a \cdot x \pmod{p} \rightarrow a = x^{-1}$$

Algorithm 10 Inversion in $\mathbb{F}_p$, taken from [5], p. 40

```
Require: Integers $a \in [0, p), p$
Ensure: $a^{-1} \mod p$
1: $u \leftarrow a, v \leftarrow p$
2: $x_1 \leftarrow 1, x_2 \leftarrow 0$
3: while $u \neq 1$ do
4:     $q \leftarrow \lfloor u/v \rfloor, r \leftarrow v - q \cdot u, x \leftarrow x_2 - q \cdot x_1$
5:     $v \leftarrow u, u \leftarrow r, x_2 \leftarrow x_1, x_1 \leftarrow x$
6: end while
7: return $x_1 \mod p$
```

The drawback of this Algorithm 10 is the multi-precision division. This division can be substituted by subtractions, which is still costly. But if we combine the subtraction with divisions by two, which are right shifts and therefore cheap, the algorithm becomes much more efficient. This can be seen in Algorithm 11. The number of iterations is at most $2k$, with k as the bitlength, in both cases, but the binary version does not contain multi-precision divisions.

Algorithm 11 Binary Inversion in $\mathbb{F}_p$, taken from [5], p. 41

```
Require: Integers $a \in [0, p), p$
Ensure: $a^{-1} \mod p$
 1: $u \leftarrow a, v \leftarrow p$
 2: $x_1 \leftarrow 1, x_2 \leftarrow 0$
 3: while $u \neq 1 \wedge v \neq 1$ do
 4:    while $u$ is even do
 5:       $u \leftarrow u/2$
 6:       if $x_1$ is even then
 7:          $x_1 \leftarrow x_1/2$
 8:       else
 9:          $x_1 \leftarrow (x_1 + p)/2$
10:       end if
11:    end while
12:    while $v$ is even do
13:       $v \leftarrow v/2$
14:       if $x_2$ is even then
15:          $x_2 \leftarrow x_2/2$
16:       else
17:          $x_2 \leftarrow (x_2 + p)/2$
18:       end if
19:    end while
20:    if $u \geq v$ then
21:       $u \leftarrow u - v, x_1 \leftarrow x_1 - x_2$
22:    else
23:       $v \leftarrow v - u, x_2 \leftarrow x_2 - x_1$
24:    end if
25: end while
26: if $u = 1$ then
27:    return $x_1 \mod p$
28: else
29:    return $x_2 \mod p$
30: end if
```

Chapter 4

Elliptic Curves

Groups, fields and finite fields have been described in general in the last chapter. This chapter will show how to define an abelian group with an elliptic curve over a field. Furthermore the underlying hard problem which can be defined for such a group is described. At the end of the chapter we take a look at the elliptic curve digital signature algorithm because it is used in practice and is the target of this thesis' work.
The underlying field for the curve can be any, but deriving the group law by means of geometry is much more comprehensible when using the real numbers as the field.
An elliptic curve E over a field $\mathbb{F}$ is defined by the Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where all coefficients a_i are $\in \mathbb{F}$ and the discriminant of the curve $\Delta \neq 0$. The last requirement assures that the curve is smooth, i.e. that there are no multiple roots in the complex numbers. Therefore there exist no points at which the curve has two or more distinct tangent lines. For details on the discriminant of the above equation see [5] p. 76.
The set of points on E/$\mathbb{F}$ is defined as all $(x, y) \in \mathbb{F}$ that satisfy the equation together with the point at infinity $\mathcal{O}$. Two examples for elliptic curves over $\mathbb{R}$ can be seen in Figure 4.1.

4.1 Simplified Weierstrass Equation

Let E_1 and E_2 be two elliptic curves in the long Weierstrass form. If there is a change of variables possible which transforms E_1 into E_2 such that

$$(x, y) \to (u^2x + r, u^3y + u^2sx + t)$$

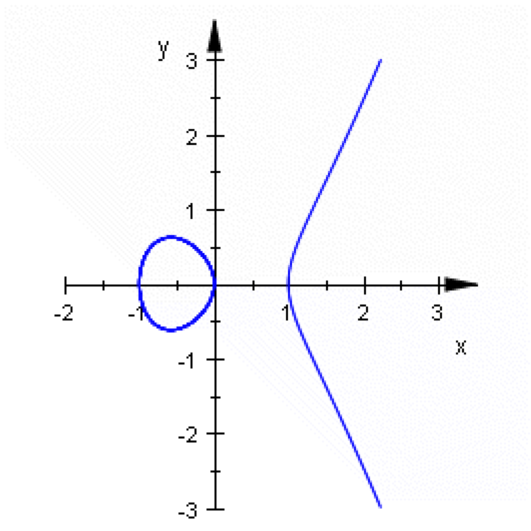
with $u, r, s, t \in \mathbb{F}$ and $u \neq 0$, then the curves are said to be isomorphic. Simply spoken, that means that the two curves are identical, apart from the representation of their points, and the transformation is bijective. Such a transformation is called admissible change of variables and can be used to simplify the equation significantly. For a curve over a prime field the following transformation can be used:

$$(x, y) \to \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24}\right)$$

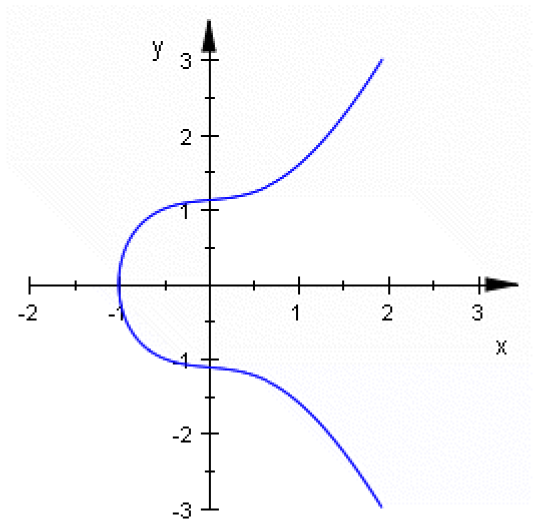
This simplifies the equation to

$$y^2 = x^3 + ax + b$$

and the discriminant of the curve to $\Delta = -16(4a^3 + 27b^2)$. For the remainder of the chapter this simplified equation will be used.



(a) $E : y^2 = x^3 - x$

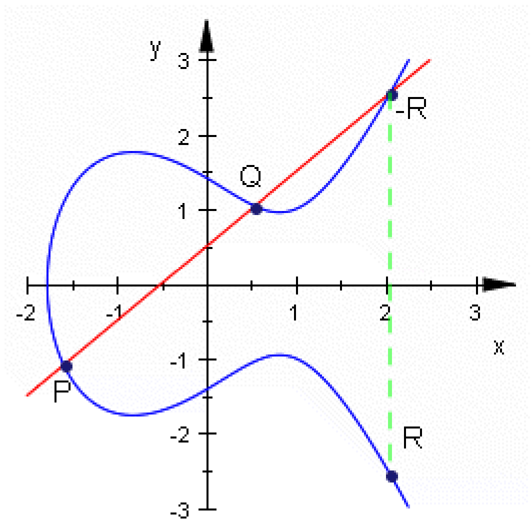


(b) $E : y^2 = x^3 + 1/4 \cdot x + 4/5$

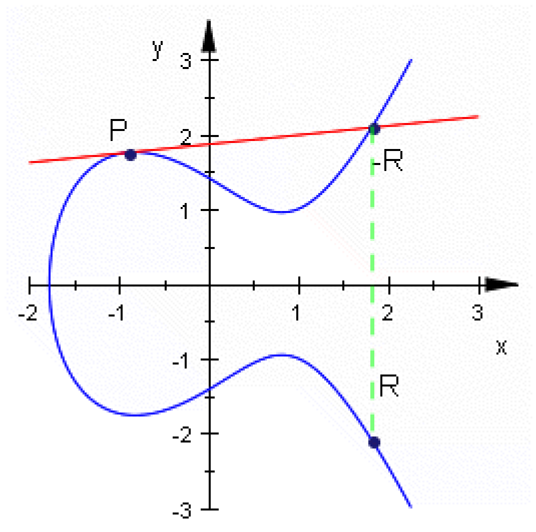
Figure 4.1: Different elliptic curves over $\mathbb{R}$

4.2 Group Law

In order to form a group there is the need for a group operation. In the case of elliptic curves this is the point addition. Adding two curve points yields a third point on the curve. The point addition itself is performed by drawing a chord through the two points and the third intersection with the curve indicates the negative of the resulting point. To get the final result, this point has to be reflected around the x-axis. If the two points are identical, a tangent has to be drawn. This can be seen graphically in Figure 4.2



(a) Addition: R = P + Q



(b) Doubling: R = 2P

Figure 4.2: Geometrical construction of the group operation

4.2.1 Chord method

In order to obtain the formula for point addition we first have to define the chord. The parameters k and d for the chord of the form $y = kx + d$ can be found as follows:

$$k = \frac{y_2 - y_1}{x_2 - x_1},\ d = y_1 - x_1 k$$

Next we have to intersect the chord with the elliptic curve. This can be achieved by solving the equation

$$x^3 + ax + b - (kx + d)^2 = 0$$

Since the left side of the equation is a third order polynomial, there have to be three solutions. Two of these solutions are represented by P and Q and are therefore already known. Hence we can divide the equation by $(x - x_1)$ and $(x - x_2)$. This yields the formula for the x-coordinate shown below. The y coordinate is easily obtained by inserting it into the curve equation.

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \tag{4.1}$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{4.2}$$

4.2.2 Tangent method

The formulas for addition cannot be applied for two identical points, since this would lead to a division by zero. However, formulas for this special case can be derived in a similar way as for addition. It is then called point doubling and represents a special case of the addition. In order to obtain the doubling formulas, we just need to differentiate the curve equation and calculate its slope at the given point. With that information and the point itself we can again define a line equation. If we then intersect this tangent with the curve equation and extract the missing root, the following formulas for the doubled point can be derived:

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \tag{4.3}$$

$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{4.4}$$

4.3 Point at infinity

Above we stated that the point at infinity is part of the curve by definition. In this section we try to look at it from a more natural point of view. For an elliptic curve there is no need to have a $\mathcal{O}$ and in fact if we look at an affine curve it is not even possible, because there exists no $\mathcal{O}$ on affine planes. However if we try to show the property of closure on affine planes there arises a problem. Assuming the doubling of a point P, where the gradient of E is ∞, there is no way to intersect the tangent with the curve again. The best case on an affine plane is that the tangent and the curve become parallel somewhere (so 2P $\notin$ E). Also the addition and doubling formulas lead to singularities in such a case.

Hence a point at infinity is necessary in order to define a group. However, to be able to introduce it, we have to map the curve on a projective plane.

4.3.1 Projective Geometry

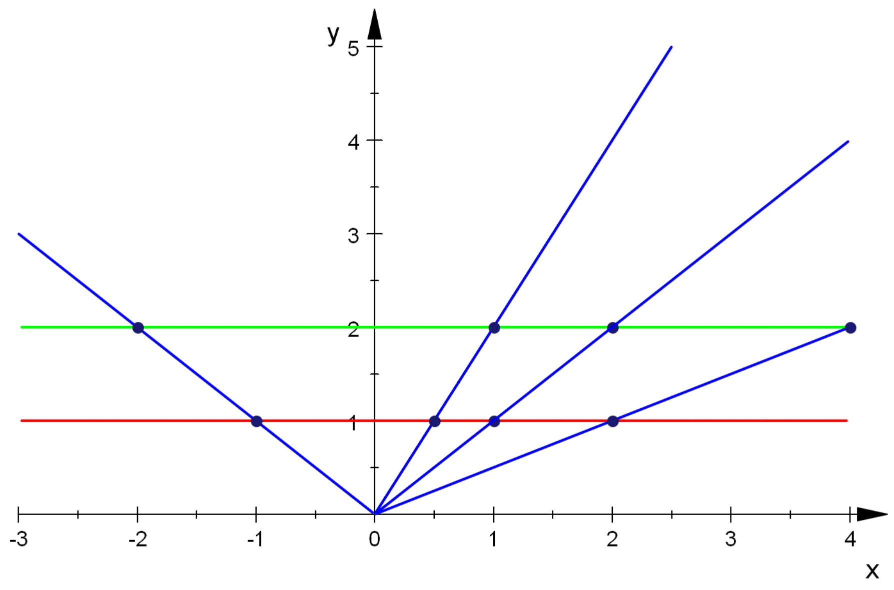


Figure 4.3: Two representatives of a projective number line with $X = x \cdot Y$. The red line is the affine equivalent. The green line is exactly the same line, but projected at $Y = 2$.

In projective geometry an object (point, line, plane,...) is defined by its equivalence class. The equivalence class has one coordinate more than the affine correspondent of the object. This extra coordinate tells us where the object has been projected. Let us take a look at Figure 4.3. It is essential to look at the whole figure as the equivalence class of a projective line and not as a plane. Hence the red and the green lines are two representatives of this projective line. The red line corresponds to the ordinary affine number line with the points $x = \{-1, 0.5, 1, 2\}$ on it. If we add a second coordinate Y and a projection rule for the x-coordinate $X = x \cdot Y$, we get the equivalence classes for the points as $(X : Y) = \{(-1 \cdot Y : Y), ..., (2 \cdot Y : Y)\}$ for $Y \neq 0$. These are the blue lines and they are indeed equivalence classes of projective points, not lines. Their representatives on the number line projected at $Y = 2$ are therefore $(X, Y) = \{(-2, 2), ..., (4, 2)\}$. The point $(X, 0)$ is the point at infinity. It is clearly part of the projective line, but it does not correspond to any affine point (this would be $(x) = (X/Y) = (X/0) = undef$).
For projective planes it works in the same way: If $\{(x, y)|x, y \in \mathbb{F}\}$ is the affine point, then $\{(X : Y : Z)|X, Y, Z \in \mathbb{F}, Z \neq 0\}$ is the projective equivalence class and the representatives within that class are defined as $\{(\lambda^c X, \lambda^d Y, \lambda Z)|\lambda \in \mathbb{F}^*\}$. Furthermore every member of the equivalence class corresponds to the affine point. The projective points for $Z = 0$ are the points at infinity and do not correspond to the affine point. The c and d of the representatives are positive integers. Depending on those two parameters different projective coordinates are obtained. For standard projective coordinates they are one, for Jacobian coordinates $c = 2$ and $d = 3$. The latter ones play a special role, because for

these coordinates the group operations are more efficient.
If we now apply those projection rules to the affine Weierstrass equation, we obtain the projective one. The curve equivalence classes can be seen in Figure 4.4 for standard coordinates and in 4.5 for Jacobian coordinates.
It has been shown that the point or line at infinity is a natural thing in projective geometry. However, why it is so important for elliptic curves in order to form a group can be explained by the second projective axiom: Given any two distinct lines, there is exactly one point of intersection. Furthermore the line at infinity is part of the projective plane. Every projective line has its own point at infinity on this line at infinity. If two lines on a projective plane are parallel, then they have the same point at infinity and thus they intersect there. In Figure 4.1 one could imagine the point at infinity (if it would exist on that plane) as the point where all vertical lines intersect. For elliptic curves on a projective plane in Jacobian coordinates (see Figure 4.5), a non trivial point at infinity would be for instance (1:1:0). In fact every point which has a Z-coordinate equal to zero and still satisfies the projective Weierstrass equation (Equation 4.5) is one. Hence after putting the curve on a projective plane, the points at infinity become automatically a part of it and close the curve.

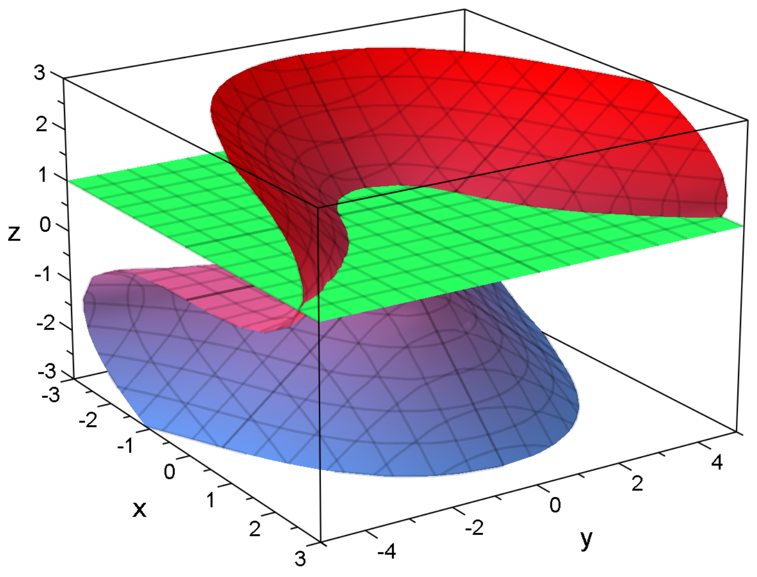


Figure 4.4: Projective curve using standard projective coordinates and its affine equivalent representation pointed out by the intersection with the plane

The projective Weierstrass equation in Jacobian coordinates is obtained by substituting x by X/Z^2 and y by Y/Z^3. After eliminating the denominators we get:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{4.5}$$

The addition and doubling formulas can be transformed in a similar manner. By substituting again and setting $Z_3 = 2Y_1Z_1$ for doubling and $Z_3 = (X_2Z_1^2 - X_1)Z_1$ for addition, the denominators can be cleared and we obtain the Equations 4.6 for doubling and 4.7 for addition.

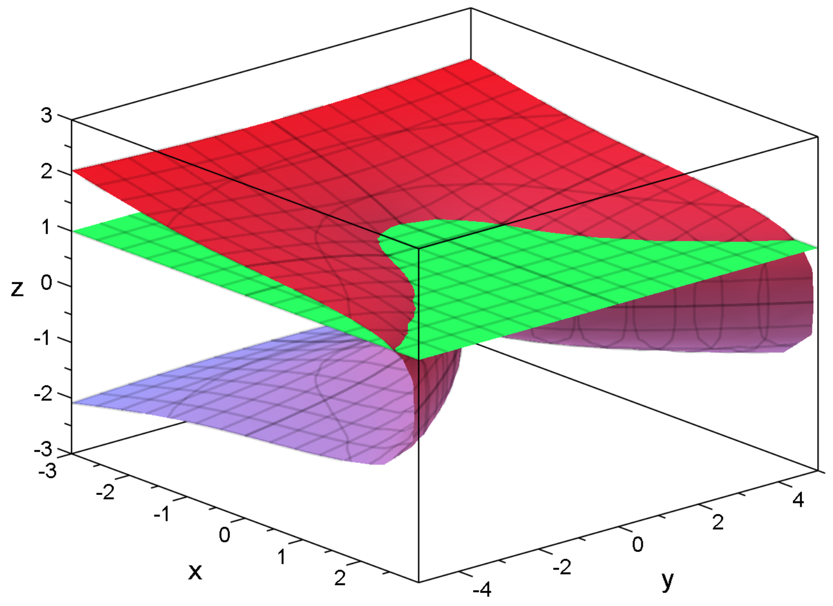


Figure 4.5: Projective curve using Jacobian coordinates and its affine equivalent representation pointed out by the intersection with the plane

$$\begin{aligned} X_3 &= (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 &= (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 &= 2Y_1Z_1 \end{aligned} \tag{4.6}$$

$$\begin{aligned} X_3 &= (Y_2Z_1^3 - Y_1)^2 - (X_2Z_1^2 - X_1)^2(X_1 + X_2Z_1^2) \\ Y_3 &= (Y_2Z_1^3 - Y_1)(X_1(X_2Z_1^2 - X_1)^2 - X_3) - Y_1(X_2Z_1^2 - X_1)^3 \\ Z_3 &= (X_2Z_1^2 - X_1)Z_1 \end{aligned} \tag{4.7}$$

4.4 Group properties

After defining the group operation and the point at infinity as the identity element, the group properties can be summed up as followed:

1. **Closure under $+$:** Since the curve together with the point of infinity forms a closed curve of order three, all tangents have to have two and all chords have to have three intersections.

2. **Identity:** The identity element is represented by $\mathcal{O}$. If we imagine the point at infinity in Figure 4.1, then there is also a nice explanation for the reflecting: $P+\mathcal{O} = P$, but drawing a tangent through $\mathcal{O}$ and P yields $-P$. Hence the resulting point has to be reflected around the x-axis in order to obtain P again. $P - P$ yields $\mathcal{O}$, which stays $\mathcal{O}$ if reflected.

3. **Inverse:** The inverse of P, $-P$, is just the point reflected around the x-axis. For curves over prime fields this can be done by taking the negative of the y-coordinate: $P = (x, y) \rightarrow -P = (x, -y)$.

4. **Addition:** The group operation for two distinct points P and Q, with $Q \neq -P$ is the addition. The double operation is just a special case of the addition for $P = Q$.

4.5 Group Operation Algorithms

Algorithm 12 Point doubling, taken from [5], p. 91

```
Require: $P = (X_1 : Y_1 : Z_1)$ in Jacobian coordinates on $E/\mathbb{F} : y^2 = x^3 - 3x + b$
Ensure: $2P = (X_3 : Y_3 : Z_3)$ in Jacobian coordinates
1: if $P = \mathcal{O}$ then
2:     return $\mathcal{O}$
3: end if
4: $T_1 \leftarrow Z_1^2$
5: $T_2 \leftarrow X_1 - T_1$
6: $T_1 \leftarrow X_1 + T_1$
7: $T_2 \leftarrow T_2 \cdot T_1$
8: $T_2 \leftarrow 3T_2$
9: $Y_3 \leftarrow 2Y_1$
10: $Z_3 \leftarrow Y_3 \cdot Z_1$
11: $Y_3 \leftarrow Y_3^2$
12: $T_3 \leftarrow Y_3 \cdot X_1$
13: $Y_3 \leftarrow Y_3^2$
14: $Y_3 \leftarrow Y_3/2$
15: $X_3 \leftarrow T_2^2$
16: $T_1 \leftarrow 2T_3$
17: $X_3 \leftarrow X_3 - T_1$
18: $T_1 \leftarrow T_3 - X_3$
19: $T_1 \leftarrow T_1 \cdot T_2$
20: $Y_3 \leftarrow T_1 - Y_3$
21: return $(X_3 : Y_3 : Z_3)$
```

This section presents the most efficient algorithms for elliptic curves over prime fields. Two facts are responsible for the performance of the algorithms. First of all we showed in the above section, that we can eliminate the denominators in the add and double formulas by introducing projective coordinates. This lets us avoid expensive inversions within the group operations. Furthermore if we use Jacobian coordinates and set the curve parameter a to -3 we can simplify the doubling equation for X_3, since $3X_1^2 - aZ_1^4 = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$. This trades 3 squarings against one squaring and one multiplication. The requirement of a = -3 is fulfilled by all curves of the NIST P class (i.e. the NIST curves over prime fields). Table 4.5 shows a comparison of needed expensive operations for different used coordinates. It can be seen that both, addition and doubling in affine coordinates use few multiplications and squarings, but need expensive inversions. Performing the same operations in Jacobian coordinates is more efficient if the the inversion is at least about four times more expensive than a multiplication. However, similar to

Algorithm 13 Point addition, taken from [5], p. 91

```
Require: $P = (X_1 : Y_1 : Z_1)$ in Jacobian coordinates, $Q = (x_2 : y_2)$ in affine coordinates
    on $\mathrm{E}/\mathbb{F} : y^2 = x^3 - 3x + b$
Ensure: $P + Q = (X_3 : Y_3 : Z_3)$ in Jacobian coordinates
 1: if $Q = \mathcal{O}$ then
 2:   return $(X_1 : Y_1 : Z_1)$
 3: end if
 4: if $P = \mathcal{O}$ then
 5:   return $(x_2 : y_2 : 1)$
 6: end if
 7: $T_1 \leftarrow Z_1^2$
 8: $T_2 \leftarrow T_1 \cdot Z_1$
 9: $T_1 \leftarrow T_1 \cdot x_2$
10: $T_2 \leftarrow T_2 \cdot y_2$
11: $T_1 \leftarrow T_1 - X_1$
12: $T_2 \leftarrow T_2 - Y_1$
13: if $T_1 = 0$ then
14:   if $T_2 = 0$ then
15:     return $(X_3 : Y_3 : Z_3) \leftarrow 2(x_2 : y_2 : 1)$ using Algorithm 12
16:   else
17:     return $\mathcal{O}$
18:   end if
19: end if
20: $Z_3 \leftarrow Z_1 \cdot T_1$
21: $T_3 \leftarrow T_1^2$
22: $T_4 \leftarrow T_3 \cdot T_1$
23: $T_3 \leftarrow T_3 \cdot X_1$
24: $T_1 \leftarrow 2T_3$
25: $X_3 \leftarrow T_2^2$
26: $X_3 \leftarrow T_2^2$
27: $X_3 \leftarrow X_3 - T_1$
28: $X_3 \leftarrow X_3 - T_4$
29: $T_3 \leftarrow T_3 - X_3$
30: $T_3 \leftarrow T_3 \cdot T_2$
31: $T_4 \leftarrow T_4 \cdot Y_1$
32: $Y_3 \leftarrow T_3 - T_4$
33: return $(X_3 : Y_3 : Z_3)$
```

the Montgomery multiplication method, the use of Jacobian coordinates only pays off if many group operations are performed. This is because the transformation back to affine coordinates is expensive. The addition becomes a bit more expensive than the doubling in Jacobian coordinates. However, the number of doublings in an efficient point multiplication algorithm is much larger then the number of additions.

Table 4.1: Number of inversion, multiplication and squaring operations for different coordinates

	Affine coordinates	Jacobian coordinates
Addition	1I, 2M, 1S	8M, 3S
Doubling	1I, 2M, 2S	4M, 4S

4.6 Point Multiplication

Until now we talked about field arithmetic, which is used to implement the elliptic curve group operations. In addition we discussed how to implement and accelerate these group operations. In this section we consider algorithms for multiple applications of the group operations. These algorithms are called point multiplication or scalar multiplication algorithms and compute kP, where k is the scalar and P is a point on the curve.

4.6.1 Binary Method

The normal binary double-and-add algorithm works analogous to a binary square-and-multiply exponentiation algorithm. We only exchange the squaring by the doubling and the multiplication by an add. Hence, we have to perform a double for every bit, plus an add if a bit of the scalar is one. It is possible to either start with the most or the least significant bit of the scalar. Algorithms 14 and 15 show both implementations.

Algorithm 14 Left to right binary point multiplication

```
Require: $k = (k_{t-1}, ..., k_0), P$
Ensure: $Q = kP$
  $Q \leftarrow \infty$
  for $i = t - 1$ to 0 do
    $Q \leftarrow 2Q$
    if $k_i = 1$ then
      $Q \leftarrow Q + P$
    end if
  end for
```

Algorithm 15 Right to left binary point multiplication

```
Require: $k = (k_{t-1}, ..., k_0), P$
Ensure: $Q = kP$
  $Q \leftarrow \infty$
  for 0 to $i = t - 1$ do
    if $k_i = 1$ then
      $Q \leftarrow Q + P$
    end if
    $P \leftarrow 2P$
  end for
```

4.6.2 Window Method

The window method can be applied if some points can be precomputed. This holds for example for the ECDS algorithm where the base point is fixed and hence also multiples of this base point are fixed. The principle behind it is quite simple. Instead of just looking at one bit, we always look at a couple of bits. Assume our window size is four, then this four bit chunk of the scalar can have 16 different values. Hence if we can precompute the 15 possible points which we have to add for each chunk, the number of additions can be reduced by a factor of four. Every addition is then followed by four doublings.

Algorithm 16 Right to left binary point multiplication

```
Require: $k = (k_{t-1}, ..., k_0), Q_0...Q_{2^w-1}, w$
Ensure: $Q = kP$
  $Q \leftarrow \infty$
  for $i = 0$ to $\lfloor (t-1)/w \rfloor$ do
    $c \leftarrow (k_i, ..., k_{i+w-1})$
    if $c \neq 0$ then
      $Q \leftarrow Q + Q_c$
    end if
    $P \leftarrow 2^w P$
  end for
```

4.6.3 Further accelerations

4.6.3.1 Non-adjacent Form (NAF)

Another method to make the point multiplication faster is to use the non-adjacent form for the scalar. In this approach the scalar is rewritten in a signed digit representation. This is useful because it reduces the length of the scalar and for elliptic curves the point negation is rather cheap. Hence point addition and subtraction is quite similar. For further information see [5], p. 98.

4.6.3.2 Shamir's Trick

With this method computing the sum of two point multiplications is faster than to compute them separately. So basically we want the result of $kP+sQ$. If we rearrange the two scalars in w bit chunks and interleave those chunks we get a new scalar of double the size and $2w$ bit window chunks. Next we can precompute all 2^{2w} combinations of $iP+jQ$ with $i = 0...2^w-1$ and $j = 0...2^w - 1$. Finally we run a normal window method. Hence the number of additions doubles, but the number of doubles stays the same. This trick can be applied in various forms. The most trivial application is for the ECDSA signature verification where exactly an expression like above appears. But also normal scalar multiplications can be brought in such a form. If we for instance know the base point in advance and the scalar has 192 bit, we can precompute $2^{96}P$ and then split the scalar to compute $k_{96}(2^{96}P)+k_0P$. However, even if the base point is not known in advance, there exist efficiently computable endomorphisms for some curves which still allow to gain speed by applying this method (see [5], p. 124).

4.7 Elliptic Curve Cryptography

In the previous sections we introduced elliptic curves, showed how they can form an additive group and described the algorithms which are used to operate on these curves. Now we want to investigate the hard mathematical problem which can be defined for elliptic curves and which is the core of all EC crypto systems. This problem is called elliptic curve discrete logarithm problem (ECDLP). Afterwards the elliptic curve digital signature algorithm (ECDSA) is presented as an example for elliptic curve based cryptographic protocols, since it is the target of the attack in this thesis.

4.7.1 ECDLP

The ECDL problem is defined as follows: Given an elliptic curve E defined over a field $\mathbb{F}_q$, a point P $\in$ E($\mathbb{F}_q$) of order n, and a point $Q \in < P >$, find the integer l $\in [0, n-1]$ such that $Q = lP$. The integer l is called the discrete logarithm of Q to the base P, denoted as $l = log_P Q$. There are several methods to break the ECDLP. However, none of the known methods runs in sub-exponential time, if the curve is chosen appropriately.

4.7.1.1 Pohlig-Hellman Attack

The Pohlig-Hellman attack follows an approach, where the discrete logarithm is solved in subgroups. The obtained system of congruences (one for each solution in a subgroup) can then be solved with the help of the Chinese remainder theorem (CRT). As stated before, the order of each subgroup has to divide the order of the group. Hence the assumption for this attack is, that the prime factors of the group order are sufficiently small. Otherwise not only solving the DLP in the subgroups becomes infeasible, but also the factorization itself can turn out to be a problem.
Given the factorization of P's order $n = \prod p_i^{e_i}$ and the points P and Q, we can first reduce the problem to one in a subgroup of order $p_i^{e_i}$ by rewriting the equation as follows:

$$Q_i = \frac{n}{p_i^{e_i}} \cdot Q = l_i \cdot \frac{n}{p_i^{e_i}} \cdot P = l_i \cdot P_i$$

If $e_i \geq 2$ we have to reduce it further by rewriting l_i as powers of p_i:

$$l_i = x_0 + x_1 \cdot p_i + ... + x_{e_i - 1} \cdot p_i^{e_i - 1}$$

By then multiplying the point by p_i, only $l_i = x_0$ remains, because all other terms are multiples of the new subgroup order. Since $x_0 \in [0, p_i)$ this equation is easy to solve by brute forcing. Next we multiply by p_i^2 and solve $l_i = x_0 + x_1 \cdot p_i$, where x_0 is already known and $x_1 \in [0, p_i)$ and so on. At the end we obtain the set of congruences $l \equiv l_i \mod p_i^{e_i}$ and solve it with the CRT.
Recapitulatory it can be said that this attack can be quite efficient and in order to prevent it, one has to make sure that the order of P has large prime factors.

4.7.1.2 Pollard's Rho Attack

Pollard's rho attack is a collision attack. It uses the following trick: If one can find a tuple (c', d', c'', d'') such that

$$c'P + d'Q = c''P + d''Q$$

then

$$(c' - c'')P = (d' - d'')Q = (d' - d'')lP$$

can be rewritten as

$$l = (c' - c'')(d' - d'')^{-1} \pmod{n}.$$

The only problem left is to find such tuples. Since we are looking for a collision, the birthday paradox can be used to approximately predict the complexity of the attack with $\sqrt{\pi n/2}$.

The single processor algorithm to find such collisions with minimal memory requirements basically defines a sequence of points X_i and computes X_i and X_{2i} until $X_i = X_{2i}$ while $c' \neq c'', d' \neq d''$. It can be found in [5], p. 159, as well as improved versions of the algorithm.

4.7.1.3 Index-Calculus Attack

This attack provides a solution in sub-exponential runtime for the discrete logarithm problem in certain groups.

Basically we define a factor base S where all primes $p \leq B \mid p \in S \subset G$. Afterwards the discrete logarithms of the elements in the factor base can be computed by factoring and solving a linear equation system. The discrete logarithm we are looking for can then be obtained by again factoring and simple substitution of the previous results. Hence the most expensive operation of the algorithm (for more information see [5], p. 165) is the factoring, which can be done in sub-exponential runtime with the number field sieve (NFS).

However this algorithm does not work for elliptic curves due to the lack of a factor base.

4.7.1.4 Isomorphism Attacks

For some elliptic curves it is possible to efficiently compute an isomorphism which maps a group generated by the curve point P to another group where the group's equivalent of the ECDLP can be solved much faster. In the worst case the number of points on the curve is prime. Hence there exists an isomorphism between $\mathrm{E}(\mathbb{F}_p)$ and the additive group $\mathbb{F}_p^+$. We can solve the discrete logarithm easily in $\mathbb{F}_p^+$ using the EEA. Attacks using the Weil pairing or the Weil descent make use of the existence of such an isomorphism. However, these attacks are not general because the isomorphism can only be computed for curves with certain parameters. Now that these curves are known, they can be avoided in cryptographic applications.

4.7.1.5 ECDLP security conclusion

In the above sections it has been stated that until now the ECDL problem can only be solved in sub-exponential runtime for certain curves. If we make sure to avoid such weak curves the runtime is exponential. However for a wide range of elliptic curve implementations the ECDL problem can be solved in linear or polynomial time by using side-channel information, see Chapter 6

4.7.2 ECDSA

The ECDSA is the elliptic curve analogue of the digital signature algorithm. The private key is a scalar d, and the public key is derived as $Q = dP$.

Algorithm 17 ECDSA signature generation, taken from [5], p. 184

```
Require: Domain parameters D = ($q$, FR, $S$, $a$, $b$, $P$, $n$, $h$), private key $d$, message $m$.
Ensure: Signature $(r, s)$
  Select $k \in [1, n-1]$
  Compute $kP = (x_1, y_1)$ and convert $x_1$ to an integer $\bar{x}_1$
  Compute $r = \bar{x}_1 \mod n$. If $r = 0$ then go back to step 1.
  Compute $e = H(m)$.
  Compute $s = k^{-1}(e + dr) \mod n$. If $s = 0$ then go back to step 1.
  return $(r, s)$
```

Algorithm 18 ECDSA signature verification, taken from [5], p. 184

```
Require: Domain parameters D = ($q$, FR, $S$, $a$, $b$, $P$, $n$, $h$), public key $Q$, message $m$,
  signature $(r, s)$.
Ensure: Acceptance or rejection of the signature.
  Verify that r and s $\in [1, n-1]$. If any verification fails then Return(“Reject the signa-
  ture“)
  Compute $e = H(m)$.
  Compute $w = s^{-1} \mod n$.
  Compute $u_1 = ew \mod n$ and $u_2 = rw \mod n$.
  Compute $X = u_1P + u_2Q$.
  If X = $\infty$ then Return(“Reject the signature“)
  Convert the x-coordinate $x_1$ of $X$ to an integer $\bar{x}_1$; compute $v = \bar{x}_1 \mod n$.
  if $v = r$ then
    return “Accept the signature“
  else
    return “Reject the signature“
  end if
```

In order to show that the algorithm works we have to show that $r = v$. This means that X needs to be kP, which holds because

$$X = u_1P + u_2Q = (u_1 + u_2d)P = kP.$$

The last step in the above equation holds because

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}rd \equiv we + wrd \equiv u_1 + u_2d \pmod{n}.$$

In order to make sure that ECDSA is GMR secure (a computationally bounded adversary who is able to mount an adaptive chosen-message attack is unable to forge a signature), the domain parameters have to be chosen in a way so that the ECDL problem is intractable and the hash function H has to be collision and pre-image secure.
If H is not collision secure the attack is trivial, we just need to find two messages which result in the same hash sum and somehow obtain a signature for one of them in order to get the other one signed. If H is not pre-image secure, we set $kP = Q + lP, s = r$ and $e = rl$. Then we need to find a message m for which H(m) is e. If this requirement is fulfilled it can be verified that

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}rd = r^{-1}(rl + dr) = l + d \to kP = lP + Q \to r = v.$$

It is also crucial that the ECDSA implementation performs all checks. Otherwise, if the check $r \not\equiv 0 \mod n$ is not done and assuming that the base point is $(0, \sqrt{b})$, an adversary can always produce the valid signature $(r = 0, s = e)$ for every m. It works because $u_1 = 1, u_2 = 0$ and hence $v = 0 = r$.
Another fatal thing to do is to reuse the ephemeral key k for two messages m_1 and m_2. In this case, subtracting the two equations $ks_1 = e_1 + dr$ and $ks_2 = e_2 + dr$ would immediately reveal k and thus d.

$$k \equiv (s_1 - s_2)^{-1}(e_1 - e_2) \pmod n$$

It is obvious that knowing k for a given signature is equivalent to knowing d. However, it is also sufficient to know only a small portion of k for many signatures. If we can collect congruences

$$e_i \equiv s_i k_i - dr_i \pmod n$$

for which e, s_i and r_i are known and the k_i's are partially known, we can use lattice attacks [6] to forge signatures. In the best case we can find find a real k_i and therefore obtain d. But even if that does not happen, there is still the chance that the found k_i makes it possible to claim, that we signed the message. The LLL algorithm used in the attack has polynomial runtime and performs the better the more bits of the ephemeral keys are know. However it is also stated in [6] that the portion of k_i, which we can reveal by our attacks described in Chapter 6 is highly sufficient.

Chapter 5

Power Analysis Attacks

This chapter will first introduce different kinds of side channels and will give a brief overview on how to use them. Then we will describe the two basic attack approaches *Simple Power Analysis* (SPA) and *Differential Power Analysis* (DPA). Later on more sophisticated variations of these two ideas, like template attacks, high-order DPA attacks and template-based DPA attacks will be presented. Then we introduce a method to combine templates which significantly increases the success probability. Finally, a technique to improve the quality of the used power traces will be shown.

5.1 Side Channels Attacks

Side channel attacks are implementation attacks. They exploit the fact that a device reveals information because of implementation characteristics, intended security performance trade offs or simply because it is not possible to disguise the leakage. Side channels are typically of physical nature, i.e. something measurable which is in some way dependent on the data. This can even be the sound produced by a desktop PC during an RSA computation [1]. The three maybe most important side channels are described in detail below.

5.1.1 Execution time

Timing attacks exploit the fact that algorithms can contain conditional execution paths like data dependent branches and loops. The data processed by cryptographic devices is a function of an input and a key. If there is a relationship between those two parameters and the execution time the algorithm reveals useful information [8][19] .
To mount such attacks accurate timing measurements and a good knowledge about the processed algorithm are needed. The aim of the countermeasures is to make the execution time independent of the data. Thus they influence the performance, because the execution time must always match with the one of the longest path.

5.1.2 Power Consumption

Power analysis attacks exploit the fact that the power consumption of a device is a function of the executed instructions, the data processed by the instructions and an internal state. Hence it contains useful information for an attacker.
To record this information one samples the power supply line of a device at a very high

frequency (several 100 MHz). The data acquired in this way is called a *power trace* and is investigated during the attacks.
There exists a huge variety of countermeasures to disguise this side channel, starting from masking the processed data in the program code to using special logic cells to build the cryptographic devices [13][3][17]. Some of the techniques work against specific attacks, but there is no technique to make a device completely resistant.

5.1.3 Electromagnetic Emanation

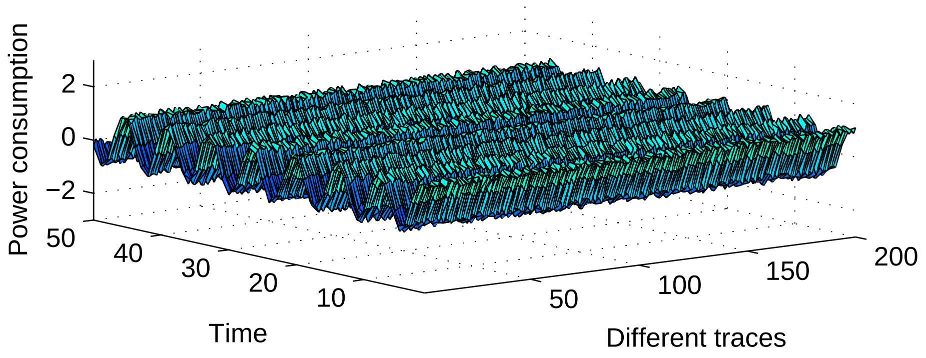
Electromagnetic analysis attacks work similar to power analysis attacks. The flowing current in the wires of a chip causes electromagnetic radiation. The information contained in the radiation can be seen analogous to the power consumption and thus be evaluated with the same methods [10]. The main difference is the data acquisition equipment.

5.2 Basic Power Analysis Methods

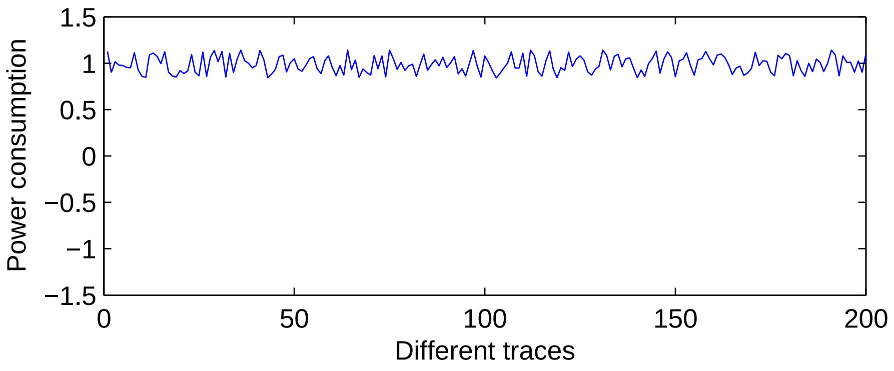
Power analysis attacks were first introduced by Kocher et al. [7] as a non-invasive method to gather information about the internals of a device. In their paper they also first defined SPA and DPA. The former one looks at one power trace along the time axis. The simplest attack could be a visual investigation in order to distinguish significant features within the trace. This could be the ability to tell apart double and add operations for an elliptic curve scalar multiplication. In such a case the key bits can be extracted without even processing the power trace in any way.
Whilst SPA uses only one trace, DPA needs several traces. It takes advantage of the fact that the same instruction dissipates different amounts of power depending on the data it processes. So we look at many traces, but at the same point within every trace. Figure 5.1 points out the different views: Figure 5.1(a) shows 200 acquired traces with 50 samples each. Let us represent this data in a 200-by-50 matrix $\mathbf{T}$, where the i-th trace is denoted by t_i and the j-th column is denoted by t'_j. If we do SPA attacks our point of view can be represented by Figure 5.1(c) which shows all 50 samples of trace t_1. For DPA attacks on the other hand we work with the information represented like in Figure 5.1(b), which shows all 200 traces at the time t=25, i.e. t'_{25}.

5.2.1 Simple Power Analysis

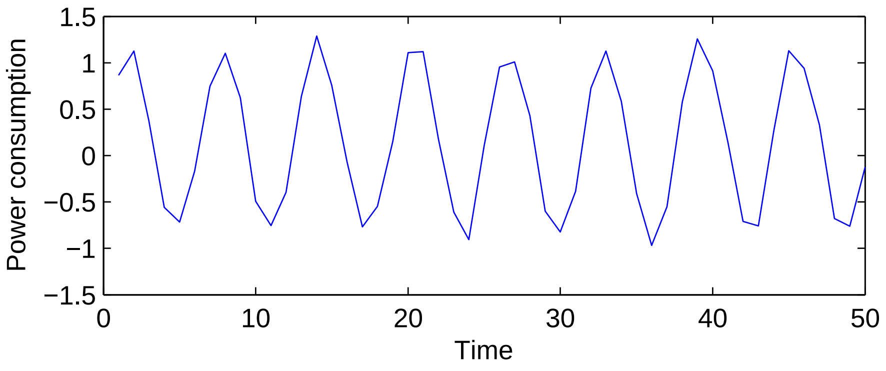
As mentioned above the simplest way of doing SPA is by visually inspecting a power trace. This can be possible in some cases like seen in Figure 5.2. It shows the power consumption of an ARM7TDMI-S core during one add and one double operation of an ECC binary left-to-right algorithm. The used algorithms need 11 multiplications for an add and 8 for an double. Thus there are 11 valleys visible during the addition and 8 during the double. Given a whole trace it is rather easy to extract the key in this way. But this works only for the simplest possible implementations. Several successful countermeasures against this kind of attack have been introduced: They unify the group operation, work with randomized addition-subtraction chains [14] or are just high performance implementations which work on several bits at once. The easiest, but also less efficient approach regarding the performance can be seen in algorithm 19 and 20. A simple modification of the double and add algorithm to a double and always add algorithm makes it resistant against this attack.



(a) All 200 traces



(b) DPA view: Same sample point for 200 different inputs



(c) SPA view: Whole execution time for one input

Figure 5.1: Different possible views on a data set

But also without countermeasures it can be quite hard to mount such an attack for pure software implementations of public-key encryption primitives. For example an ECC add or double operation for the P192 curve lasts for ten-thousands of cycles on our ARM core. In the above example it was only possible to find patterns using visual investigation after clocking the chip at the lowest possible frequency and using pure assembler code for the field arithmetic.

Nevertheless the property of SPA attacks that we only need one trace is highly interesting if the number of available traces is restricted. Imagine an attack on the ECDSA algorithm (see Algorithm 17). Here an ephemeral key is used for the point multiplication. If this key can be extracted, the private key can be as well, but the attacker has only one chance to do this. To successfully mount an SPA attack even if there are countermeasures more sophisticated methods like template attacks are needed. We will take a look at them later in Section 5.3.1.

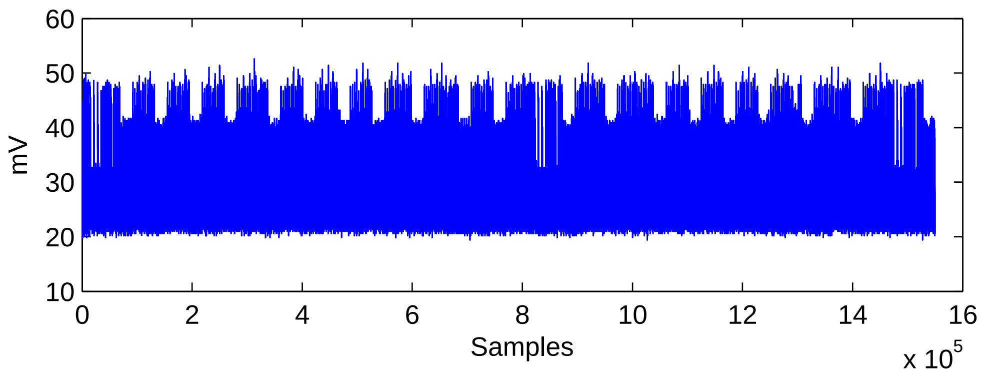


Figure 5.2: Power trace for an ARM7 during one add and one double operation

Algorithm 19 Left to right binary point multiplication

```
Require: $k = (k_{t-1}, ..., k_0), P$
Ensure: $Q = kP$
  $Q \leftarrow \infty$
  for $i = t - 1$ to $0$ do
    $Q \leftarrow 2Q$
    if $k_i = 1$ then
      $Q \leftarrow Q + P$
    end if
  end for
```

Algorithm 20 Uniform left to right binary point multiplication

```
Require: $k = (k_{t-1}, ..., k_0), P$
Ensure: $Q_0 = kP$
  $Q_0 \leftarrow \infty$
  for $i = t - 1$ to $0$ do
    $Q_0 \leftarrow 2Q_0$
    $Q_1 \leftarrow Q_0 + P$
    $Q_0 \leftarrow Q_{k_i}$
  end for
```

5.2.2 Differential Power Analysis

The principal behind DPA attacks is to look at a large amount of power traces and to find out the key all traces have in common. Most cryptographic algorithms take an input value and a key. The intermediate values processed in the algorithm are dependent on these values. So if we know the input values and guess the key, we can compute the intermediate values and check if our predicted values occur in the traces. This is possible since in Figure 5.1(b) the differences in the power consumption are dependent on the processed data and not on the instruction.

The following way of looking at a DPA and also the used notation is taken from [11]. The attack consists of several steps: First we have to find a function or a specific intermediate value to attack. Afterwards we acquire traces for which we know the input values. Then we predict the intermediate values with guessed keys and compute the predicted power consumption for our key hypothesis. In the last step we compare the prediction and the power traces by means of statistics.

5.2.2.1 Function to attack

First of all we need a point in the algorithm for which we can compute the intermediate value that depends on the key. The intermediate values to attack are denoted by the matrix **V** where each element $V_{i,j}$ is the result of a function f(d_i,k_j) and d_i is a known input data value. For the key k_j we have to guess several different values. For every guessed value we get a different intermediate value hypothesis vector v'_j. An Example for the function f could be a non-linear mapping g of the bit-wise exclusive-or-ed operands so that $V_{i,j} = \mathrm{g}(d_i \oplus k_j)$.
The non-linearity of the chosen function is quite important. In most applications the power consumption depends on the Hamming weight of the processed intermediate values. If we now only take a look at the result of a multiplication then we can hardly distinguish the Hamming weight of $1 \cdot d_i$ from the Hamming weight of $2 \cdot d_i$. If the bits are uniformly distributed than the result does not differ at all in half of the cases and only differs by one in all other cases. Or in other words, if the function we attack is not non-linear enough, then the correlation between the different hypotheses becomes high and therefore the needed trace quality to determine the right hypothesis increases dramatically.

5.2.2.2 Data acquisition

Now we execute the algorithm on our device and record its power consumption for each of the D entries in d. After that we have a matrix of power traces **T**, where each row t_i corresponds to d_i. Each column t'_j corresponds to a moment in time and one or more of them correspond to v'_j for the right guessed k_j.

5.2.2.3 Using a power model

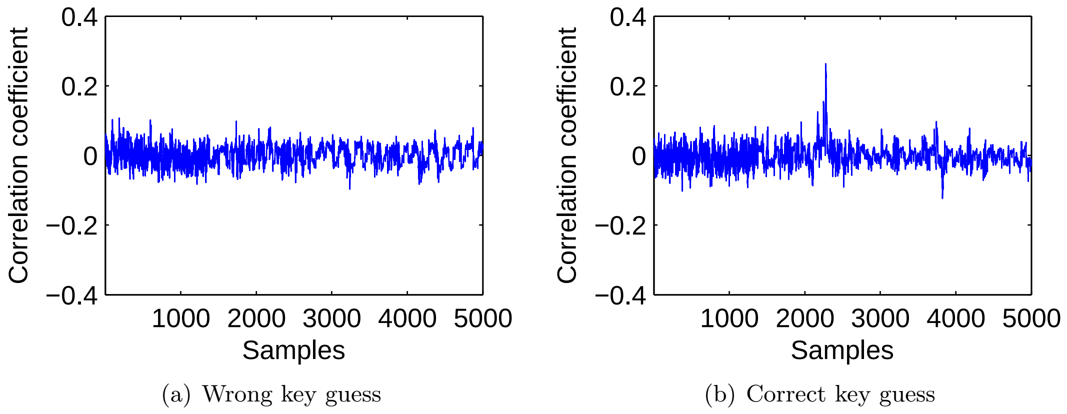
After having computed the power traces and intermediate value hypothesis vectors we want to somehow compare them. Therefor we have to map v'_j to a power consumption hypothesis h'_j. This can be achieved via a suitable power model for the device. That could be for example the Hamming weight model, where we assume that the power consumption is directly related to the number of ones in the binary representation of the data word. As shown in Section 2.3.1 this model is perfectly suited for the LPC2124 processor.
Other power models could be a simple bit model, a hamming distance model or a zero-value model. The first one works like the Hamming weight model, but only considers one bit. The hamming distance model assumes that the power consumption correlates with the difference between two occurring hamming weights. The last one exploits a special behavior of some devices if they process zero values.

5.2.2.4 The correlation step

In the last step of a DPA we want to know which t'_i for $1 \leq i \leq K$ and which h'_j $1 \leq j \leq L$ are related to each other. So we correlate each column of the power consumption hypothesis matrix **H** with each column of the trace matrix **T** and get the K-by-L matrix **R**:

$$R_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i}) \cdot (t_{d,j} - \bar{t_j})}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i})^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t_j})^2}}$$

For a high correlation we can look at i and j where the value of j tells us the used key and the value of i tells us the position where the intermediate value was processed. In Figure 5.3 you can see two correlation results. Figure 5.3(a) corresponds to the result for a wrongly guessed key and therefor contains only some minor peaks. Whereas Figure 5.3(b) contains a significantly higher peak since here the hypothesis was correct.



(a) Wrong key guess (b) Correct key guess

Figure 5.3: DPA result for different hypotheses

5.3 Advanced Power Analysis

In Section 5.2 we discussed basic DPA and SPA techniques. It has also already been stated that basic SPA attacks do not work in most cases. Furthermore there exist efficient countermeasures against normal DPA attacks (first-order DPA attacks). This section will present four more sophisticated methods which have been developed.

5.3.1 Template Attacks

Template attacks consist of two phases: In the characterization phase statistical methods are used to describe the power consumption characteristics of specific operations. The aim of the second phase, the matching phase, is to classify parts of a power trace. This phase's result tells us the probability that a specific part of a trace matches with a template. In this section an operation denotes one or more instructions with specific processed Hamming weights.
First we take a look at the different components of such an operation. Then the multivariate normal (MVN) distribution, which represents the core tool for templates, will be described. Afterwards the different stages of a template attack will be shown in detail and at the end point selection methods will be introduced.
An operation is represented by several sampled points. For each point, 3 different components contribute to the overall power consumption. These are:

- P_{el} denotes the electronic noise, which is normal distributed and contains no information, so it can be neglected.
- P_{sw} denotes the switching noise, it results from the power consumption of CPU parts that do not provide information. In our case, these are the first and the second pipeline stage and some peripherals. It also might be the case that we do not exploit all available information. If we for example use a bit model on a 32-bit processor, we neglect 31 bits of information and therefore they contribute to the switching noise, whereas they contribute to P_{exp} in a 32-bit Hamming weight model.
- P_{exp} denotes the exploitable power consumption. It is data and instruction dependent. The instruction dependent part can be seen as normal distributed, although this can differ from device to device. The data dependent part is a superposition of several distributions: First the power consumption for every specific Hamming weight is normal distributed. Additionally the occurring Hamming weights are binomial distributed if the underlying bits are uniformly distributed. The overall distribution can be elaborated by superposing the weighted normal distributions for all Hamming weights.

It is obvious that P_{el} and P_{sw} are not of interest for templates. Since the Hamming weight of P_{exp} is constant for one operation, there is only a normal distribution left. So if we have only one template which consists only of one point, we can describe it with one normal distribution. Let us assume we have 100 traces where there occurs a store instruction with Hamming weight 16 at point 50. We just estimate the mean μ and the variance σ^2 to define our normal distribution. That is all we have to do in the characterization phase. In the classification stage we take point 50 of our test trace and evaluate the probability density function (PDF).

$$p(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The closer the point of our test trace is to μ the higher the probability becomes. The highest possible value is $1/(\sigma\sqrt{2\pi})$.
Of course we record several points per instruction, so we want to have a more accurate template which takes several points into account. If we assume that all points are independent from each other we could just multiply the PDFs. Here is an example for N points:

$$p(X;(\mu_1,\ldots,\mu_N),(\sigma_1,\ldots,\sigma_N)) = \frac{1}{\sigma_1\sqrt{2\pi}}\cdots\frac{1}{\sigma_N\sqrt{2\pi}}\, e^{-(\frac{(x_1-\mu_1)^2}{2\sigma_1^2}+\ldots+\frac{(x_N-\mu_N)^2}{2\sigma_N^2})} \tag{5.1}$$

Now we can rewrite the equation with the vectors X and m and the matrix Σ' which contains the variances in its diagonal:

$$p(X;m,\Sigma') = \frac{1}{(2\pi)^{N/2}\left|\Sigma'\right|^{1/2}}\exp\left(-\frac{1}{2}(X-m)^\top\Sigma'^{-1}(X-m)\right)$$

The only thing which we still did not consider is that the points are not independent. So instead of the variance matrix Σ' we have to use the covariance matrix C. From now on we will use the notation for traces from Section 5.2.2 again. If we have a K-by-L trace matrix **T**, where the rows are the variables and the columns are the observations, we can compute the mean vector **m** and the covariance matrix **C** with

$$\mathbf{m} = \frac{1}{K}\sum_{i=1}^{K} t_i \quad \textit{and} \quad \mathbf{C}_{i,j} = \frac{1}{K}\sum_{i=1}^{K}\sum_{j=1}^{K}(t'_i - \bar{t'_i})^\top \cdot (t'_j - \bar{t'_j}).$$

The result is the PDF of an MVN distribution:

$$p(t;(\mathbf{C},\mathbf{m})) = \frac{exp(-1/2(t-\mathbf{m})^\top C^{-1}(t-\mathbf{m}))}{\sqrt{(2\pi)^K det(\mathbf{C})}}.$$

To understand how templates work it is necessary to know how **m** and **C** work together. The mean vector describes the basic power consumption level of an operation. Whereas the covariance matrix describes the variance of each variable in the diagonal and the dependencies between the deviations of the different points in the remaining elements. The more dependencies there are between the points, the larger the non-diagonal entries of the covariance matrix are. Figure 5.4 and the following covariance matrix illustrate this behavior for simulated power traces:

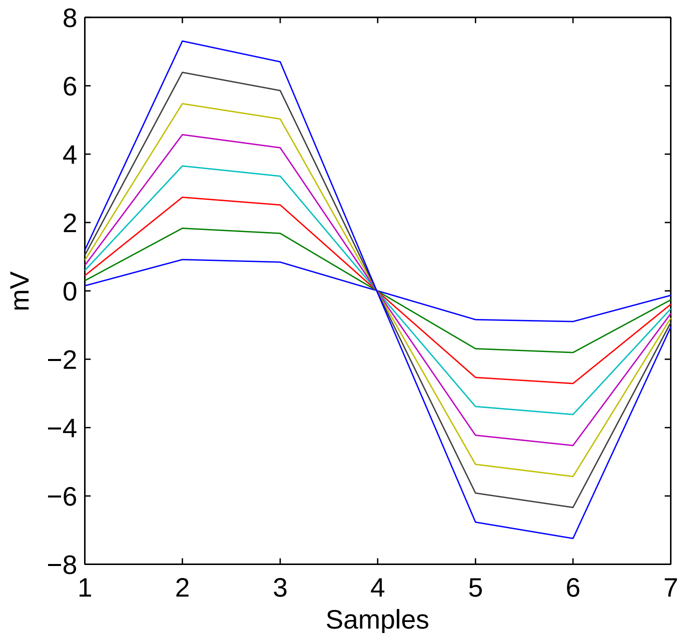


Figure 5.4: Centered simulated power traces

$$\mathbf{C} = \begin{bmatrix} 0.134 & 0.818 & 0.750 & -0.008 & -0.759 & -0.812 & -0.119 \\ 0.818 & 4.999 & 4.583 & -0.046 & -4.633 & -4.961 & -0.727 \\ 0.750 & 4.583 & 4.202 & -0.042 & -4.248 & -4.548 & -0.667 \\ -0.008 & -0.046 & -0.042 & 0.000 & 0.043 & 0.046 & 0.007 \\ -0.759 & -4.633 & -4.248 & 0.043 & 4.294 & 4.598 & 0.674 \\ -0.812 & -4.961 & -4.548 & 0.046 & 4.598 & 4.922 & 0.722 \\ -0.119 & -0.727 & -0.667 & 0.007 & 0.674 & 0.722 & 0.106 \end{bmatrix}$$

The shown traces consist of 7 samples each, where point 2, 3, 5 and, 6 have the highest variance and have pairwise a high covariance, since 2 and 5 always change in the same manner like 3 and 6 do. Furthermore the matrix describes the reciprocal proportional behavior between 2, 3 and 5, 6. This can be seen by the large negative values in the matrix. So the covariance matrix can be seen as a pattern for dependent random variables (If point 2 deviates from the mean by x then 3 should deviate by y and 5 by -z... for each variable).
If we acquire traces for exactly the same processed data value, then the differences between the traces are only caused by electrical noise. Therefore the covariance matrix contains no useful information. For such cases it is better to use the reduced templates shown in Section 5.3.1.3. But if we acquire the traces for the same Hamming weights then the traces differ because their processed values differ and therefore the covariance matrix's entropy is not zero.

5.3.1.1 Template building phase

In order to characterize operations with templates, we first need to find the points in the traces, which contain the most information. Two approaches to accomplish this task are presented in the Sections 5.3.1.5 and 5.3.1.6. After reducing the traces to some interesting variables per observation we can start to build the templates.
Above we only calculated one mean vector covariance matrix pair $(\mathbf{m}, \mathbf{C})$, i.e. only one template. So the only information we got was how good a trace matches the template. But actually to classify a trace we need to know which template the trace matches best. Hence we need one template for each operation. Thus we group the previously acquired traces into i operations and for each group we compute $(\mathbf{m}_i, \mathbf{C}_i)$.

5.3.1.2 Template matching phase

In the template building phase we characterized the MVN distribution for every different operation. In order to classify a trace, we evaluate the PDFs of all operations with a given trace. We get the probability that a trace matches for operation i with $p(t; (\mathbf{C}_i, \mathbf{m}_i))$. After we computed the probabilities for all operations we decide for the one with the highest. This is called maximum-likelihood decision rule.

5.3.1.3 Reduced Templates

Sometimes it would be quite convenient to omit the covariance matrix. It often causes numerical problems since it is badly shaped and close to singularity. We already stated that if the templates are build for specific data values and not for Hamming weights, than the variations of the traces contain no useful information. Hence it is not possible to characterize them in a meaningful way and it can only make the result worse. Instead of the covariance matrix, the identity matrix can be used within the PDF. In this case we only compute the least mean square.

5.3.1.4 Optimization

If we rewrite the above equation as

$$f(x) = \frac{exp(-(1/2) \cdot \mathbf{x})}{\sqrt{(2\pi)^K det(\mathbf{C})}}$$

we can see that it is monotonic and reciprocal proportional to $\mathbf{x}$, so we only need to compute the vector-matrix-vector product and choose the smallest resulting absolute value.

5.3.1.5 Point selection with a DPA

This point selection method first applies a DPA on the traces. Since the labels (the Hamming weights) of the trainings traces are known, it is easy to build the hypothesis. The result of the DPA shows peaks where the signal to noise ratio is high according to the information we modeled the hypothesis for. The only remaining task is to find out how many points and which points to chose.
In [18] it is proposed to chose no more than one point per clock cycle and only points which guarantee a sufficient high signal-to-noise ratio. If the templates are built for only one intermediate value this approach only minimizes the needed heuristics. Whereas if we attack enough intermediates at once we can get rid of the heuristics by just taking the best DPA point for every intermediate. This can be applied if we for example build templates for points on an elliptic curve, see Section 5.3.2.

5.3.1.6 Building a subvectorspace with PCA

The method of Principal Component Analysis (PCA) comes from the computer vision sector and was proposed by Archambeau et al. [2] to enhance template attacks. In opposite to the DPA based template attacks, this approach can be theoretically optimal in some cases without any need for heuristics. PCA finds orthogonal basis vectors for a given vectorspace, where the new axes have the property that their directions correspond to the directions of the highest variances of the data set. Furthermore there is a metric which tells us, which basis vectors are most important, i.e. represent the directions of the largest variances. These vectors are called principal components. And this points out the advantage of a PCA: Assuming that the points which contain most information are those with the highest variance, we can use a small amount of basis vectors to describe the new vectorspace without significant loss of information.
The assumption above can be easily met. If we average the traces for every Hamming weight, than they should only differ in those points, that depend on the Hamming weight. Hence if we apply a PCA on these mean vectors, we get our coordinate system. But here arise two problems: First, if we want to apply the PCA on all points of a traces, we need sufficient many traces in order to get rid of the noise, i.e. the data independent variances between the mean vectors. If we do not have enough traces, we have to restrict the interval for the PCA which needs heuristics again. Second, if we have for example only 33 different operations, e.g. 33 different Hamming weights, then we get only 33 orthogonal basis vectors for our new subvectorspace and that might not be enough to describe a reasonably accurate subspace. A way to get more components is to divide the set for the mean vectors and to use two sets for every Hamming weight to get 66 components. This increases the accuracy of the new subvectorspace, but also reduces the quality of the principal components. So again, there are some heuristics needed.

The PCA itself is a linear transformation, so the new basis vectors are weighted sums of the old ones. The weight can be found as followed: Let us assume a matrix **T** were the rows are the centered mean vectors for the different operations. In order to perform a PCA we just need the covariance matrix of all the variables in **T** and compute the eigenvalues and eigenvectors. The former ones represent the metric we mentioned above and the latter ones are the basis vectors. But now there arises a problem: **T** is a N-by-M matrix where N is the number of different centered mean vectors and M is the number of samples. The covariance matrix **S** is

$$\frac{1}{K} \cdot \mathbf{T}^\top \cdot \mathbf{T}$$

which means that its dimension is M-by-M. First, this is normally infeasible to compute and second, the rank of the matrix will only be the number of mean vectors N minus one. Fortunately there exists a trick with which we can compute the first N-1 eigenvectors with much less effort. Instead of computing **U**, the eigenvectors, and Δ, the diagonal matrix of eigenvalues, for

$$S \cdot U = U \cdot \Delta$$

we use **S'** which is

$$\frac{1}{K} \cdot \mathbf{T} \cdot \mathbf{T}^\top .$$

This is much easier and results in a N-by-N matrix. By now left multiplying $T^\top$ we get

$$S \cdot \mathbf{T}^\top U = \mathbf{T}^\top U \cdot \Delta .$$

Now we can compute the normalized principal components

$$V = \frac{\mathbf{T}^\top U}{\sqrt{N \cdot \Delta}}$$

and take an arbitrary number of columns to describe the transformation matrix in our new subspace. The importance of the vectors regarding the variance is described by the eigenvalues. The only thing left to do, is to map every trace into the new space by left multiplying it with the transposed columns.

5.3.2 Templates for high-level Operations

For some attacks a classification performance of almost 100% is not only desired but necessary. Assume an on-the-fly template attack on an 192 bit ECDSA. On-the-fly means that we do not generate 2^n templates for all possible values of the n bits of the ephemeral key in advance, but only two at once for every of the n steps. After the first i bits have been attacked and are known, two templates for bit i+1 are build and so on. Two conclusions can be drawn from this:

1. We only have to decide between two templates. That makes it easier, particularly if the difference between the Hamming weights for which the two templates were built is large. But still, the classification performance can be way below 100%.

2. If one guess is wrong the whole attack fails, because there is no metric that tells us which bit is likely to have been guessed wrong. Furthermore even if the overall success probability for a 192 bit private key should only be 0.5 and if we consider lattice attacks as well [6], we still need a template success probability of $0.5^{1/10} = 0.9330$.

Fortunately we can extract way over 100 intermediate values for an ECC double and add operation. If every intermediate value will be attacked with a template and we combine enough results a rather small success probability for each single template is sufficient. Theoretically it only has to be greater than 50% if enough intermediates are available. Assume N templates and a template success probability of p. Then we succeed if more than N/2 templates vote for the right bit. The probability that this happens can be obtained by evaluating the cumulative binomial probability function

$$\Pr(X > N/2) = \sum_{j=\lfloor N/2 \rfloor + 1}^{N} \binom{N}{j} p^j (1-p)^{N-j}.$$

In Figure 5.5 a template success probability of only 75% was assumed. Nevertheless with 50 templates the success rate reaches 99.992%. The graph shows how the success probability for the guess of one bit increases with the number of used templates.

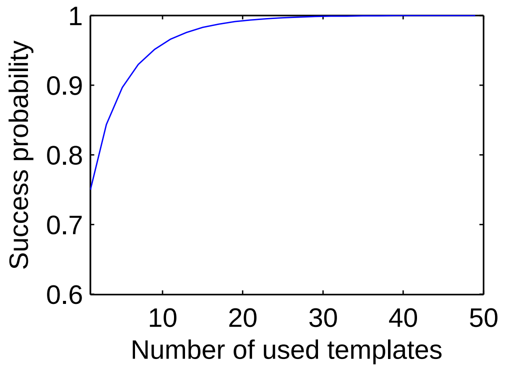


Figure 5.5: Bit guess success rate for multiple templates

Experiments have shown that neglecting the numerical template results and only counting the votes influences the result negatively. Fortunately the intermediates are independent enough so that we can just multiply the template results. Since we use reduced templates this can be seen as building one large template.

5.3.3 Template based DPA

A normal DPA uses a power model like the bit model, the Hamming weight model or the hamming distance model. The better the power model we use, the better are the results of the DPA. Furthermore templates are the most precise description of the power consumption. Hence template based DPA attacks represent the strongest form of a DPA attack.

The attack works a bit different from normal DPA attacks. Instead of correlating with every power consumption hypothesis, we map the intermediate values to templates and evaluate the templates for all traces. The joint probability of the templates for the correct hypothesis should be the highest and thus indicate the correct key.

Basically we want to know the probabilities for the different keys, given different traces which used the same key. So the information we look for is $p(k_j|\mathbf{T})$. This can be calculated with Bayes' theorem:

$$p(k_j|\mathbf{T}) = \frac{\left(\prod_{i=1}^{N} p(t_i|k_j)\right) \cdot p(k_j)}{\sum_{l=1}^{K} \left(\prod_{i=1}^{N} p(t_i|k_l)\right) \cdot p(k_l)}$$

This equation results from iteratively applying Bayes' theorem. First we start with trace one of $\mathbf{T}$:

$$p(k_j|t_1) = \frac{p(t_1|k_j) \cdot p(k_j)}{\sum_{l=1}^{K} p(t_1|k_l) \cdot p(k_l)}$$

Where the prior probability of the keys is 1/K in this case since we assume that all keys are uniformly distributed. Hence the sum over all k_l is 1. The probabilities $p(t_i|k_j)$ and $p(t_i|k_l)$ are the results of PDFs from the templates. As a result we get the prior probabilities for the keys for trace two ($p(k_j)$ and $p(k_l)$)

$$p(k_j|t_2) = \frac{p(t_2|k_j) \cdot p(k_j|t_1)}{\sum_{l=1}^{K} p(t_2|k_l) \cdot p(k_l|t_1)}.$$

5.3.4 High Order DPA

The last method we take a look at is the high order DPA. It is a DPA on the joint power consumption of several points and is used to attack implementations which are resistant against first-order DPAs like discussed in Section 5.2.2. They were already mentioned in [7] and have first been successfully applied by Messerges [12].
First-order DPA resistant implementations use arithmetical or Boolean masking to blind the intermediate values during the algorithm. The arithmetical approach uses addition or multiplication as the combination operation. For Boolean masking the intermediate values and the masks are combined using the bit-wise exclusive-or operation. We will focus on second-order DPA attacks for the latter masking method like elaborated in [15].
The difference between a first- and a second-order attack is the trace preprocessing step. Afterwards a normal DPA can be applied. So the main question of this section is how to preprocess the traces.
If only one bit intermediate values are processed then it is possible to fully recover the power consumption:

Table 5.1: Relation between the intermediates, the masks and their Hamming weights

Value v	Mask m	v $\oplus$ m	$\|v - m\|$	HW(v $\oplus$ m)	|HW(v)-HW(m)|
0	0	0	0	0	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0

Table 5.1 clarifies that the absolute power consumption of the difference between the point where the data value is processed and the one where the mask is processed correlates perfectly with the power consumption of the masked intermediate value. However this holds only for one bit. If the word size becomes larger the correlation between the last two columns in the table decreases. Experiments have shown that it is approximately 0.24 for

8 bit and 0.115 for 32 bit.
The only thing left to do is to compute the joint power consumption. Since we do not know when the mask and the intermediates are processed we have to pairwise subtract all points from each other. One can easily see that the size of the traces increases significantly to $N * (N-1)/2$ if N was the original trace size.

5.4 Noise reduction

The performance decreasing component of power traces is the noise. The higher the noise, the lower is the signal-to-noise ratio and consequently more traces are needed for a successful DPA and template attacks can become infeasible. Of course the noise can be reduced by just averaging many observations for the same input, but this requires more power traces and is not always possible. Instead we can preprocess each trace for its own. The basic assumptions which the following two approaches have in common are that the noise is normal distributed and of course independent of the signal.
Both of them work with windowing methods, which means that the information of a specific interval within the trace is compressed to one point. That does not really matter for DPA attacks if the window size is reasonable, but it can reduce the performance of template attacks.

5.4.1 Power method

For the power method we compute average power consumption for a window. If the signal is S(t) and the noise B(t) and the overall power consumption is O(t) = S(t) + B(t). The average power consumption for a window of size N starting at point t is then:

$$\begin{aligned} P(t) &= 1/N \sum_{i=t}^{t+N-1} (S(i) + B(i))^2 \\ &= 1/N \sum_{i=t}^{t+N-1} S(i)^2 + B(i)^2 + 2 \cdot S(i) \cdot B(i) \\ &= 1/N \sum_{i=t}^{t+N-1} S(i)^2 + B(i)^2 \end{aligned}$$

Since the last term in the second line is the covariance of to independent variables it is 0. The average power of the noise should basically be the same for every t and so the noise reduces to a constant value that does not influence the information in the signal.
Additionally to the noise reduction this method can also be used for compression, for window step sizes greater than one the trace length decreases, but also the information can decrease dramatically for too large step sizes.

5.4.2 4th order cumulant method

This method uses high order statistics to suppress the noise [9]. It needs the additional assumption that the signal is super-Gaussian. It was already stated above that a normal distribution can be characterized by its mean and its variance. The higher order moments like the skewness (3rd order) and the kurtosis (4th order) are zero. For a super-Gaussian distribution the kurtosis is positive.

Cumulants are functions of the moments up to the order of the cumulant. The 4th order cumulant has the interesting property that it suppresses Gaussian noise components of the signal and is defined as

$$\kappa_4(x) = \mu_2(x) - 3 \cdot \mu_2(x)^2$$

Its estimator for a window of size N starting at point t can be computed as follows:

$$\hat{k}_4(t) = \frac{N+2}{N \cdot (N-1)} \sum_{i=t}^{t+N-1} O(i)^4 - \frac{3}{N \cdot (N-1)} \left(\sum_{i=t}^{t+N-1} O(i)^2 \right)^2$$

5.4.2.1 Finding the window size

Unlike the power method, there exists an estimator for the optimal window size. It was mentioned above that the 4th order cumulant needs a high kurtosis value in order to work well. Further the kurtosis value varies with the window size. So before we start to preprocess the power traces we only take a look at one trace and compute the kurtosis values for different window sizes. This can be done by dividing the 4th order cumulant by the square of the variance variance. The resulting curve will have an absolute maximum at the beginning and a local maximum later on. The window size for which the local maximum occurs is the one we are interested in.

Chapter 6

Side-Channel Attacks on ECDSA

In this chapter we present the core work of the thesis. We elaborated and mounted several attacks on the ECDS algorithm. The goal of all these attacks is to extract the ephemeral key k and the private key d respectively. First we will describe how to mount a DPA attack on the algorithm. However, the main part of the chapter will explain the single shot SPA attacks, namely template-based SPA attacks, which we implemented. At the end we will talk about the effect of various countermeasures on the attacks.

6.1 DPA Attack

In the ECDSA signature generation algorithm (Algorithm 17) the last computation presents a suitable target for DPA attacks. Let us recall that the portion s of the digital signature (r, s) is computed by

$$s = k^{-1}(e + dr) \mod n.$$

The value of r is known and d is the private key which we want to extract by the attack. Next we take a look at the multiplication algorithm for long integers (Algorithm 5). It can be seen that with the partial knowledge of the operands, parts of the result can be calculated. More precisely, if we know the m least significant words of operand a and the n least significant words of operand b, we can calculate the $m + n - 1$ least significant words of the result c.
From the formula for s it follows that the result of $d \cdot r$ appears unreduced within the computation. At least every word of the unreduced result can be detected once by means of DPA. This holds independent of the used multiplication algorithm. Since r is known, we could guess word by word of d to build a DPA hypothesis. However, guessing 32 bit at once is infeasible. Therefor we made some experiments on how many bits are needed in order to be able to verify the hypothesis. It has turned out that 8 bit are sufficient. This amount is still small enough to make guesses feasible. The previous observations lead to the following strategy:

1. Let r be m bytes long and assume that the bytes $d_{i-1}...d_0$ for $i = 0...m-1$ of the private key d are already known. First we compute the 256 different results for all possible d_i. Hence the results differ for the bytes $m+i-1$ and $m+i$, but the latter one is not complete yet and does not appear like this in the full result.

2. We take byte $m + i - 1$ and build the hypotheses for it. Then we correlate our hypotheses with the power consumption while the word $\lfloor (m+i-1)/4 \rfloor$ is processed.

3. We vote for the partial key suggested by the hypothesis with the highest correlation. After fixing that byte, we start again with step 1 until the whole key is recovered.

However, there is a problem with this strategy. We already stated in Section 5.2.2 that the non-linearity of the attacked function is of great importance. In our case the non-linearity for the first 8 bit we guess is not given at all. This is because our hypotheses only depend on the last byte of r and the assumed part of d. If we look at two hypotheses for d_0 equal $00000001b$ and $00000010b$, the correlation coefficient between these two hypotheses itself is already 0.92. Furthermore we do not have the whole information about the intermediate value we attack, but only a quarter of it. Hence it is very likely that the correlation indicates a wrong positive for our partial key guess.
The following experiment will confirm this thesis. We generated 1000 values for r and a fixed d. Then we build the 256 hypotheses and correlate them with the simulated power values for the right correct d. Be aware that in this scenario the simulated power values reflect a perfect Hamming weight model without noise. Hence in reality, the factor of randomness can make the result even worse.

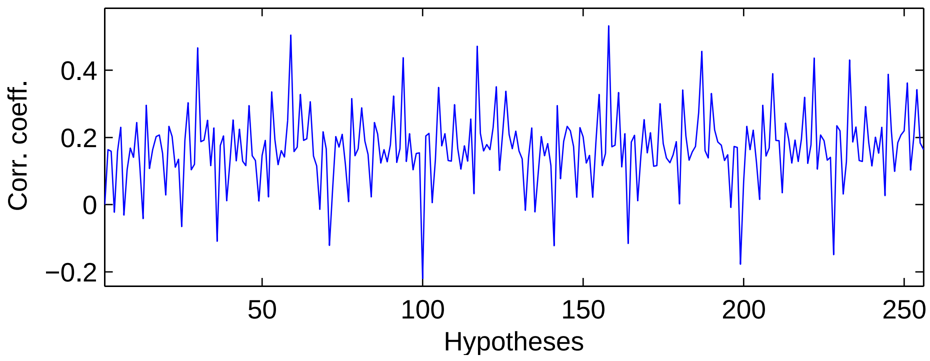


Figure 6.1: Correlation between the hypotheses for the first byte of d and the simulated power consumption for a 32-bit memory transfer.

It can be clearly seen in Figure 6.1, that the right partial key $116d$ has only the third highest correlation coefficient and $157d$ is indicated as the correct key. However, if we take a look at d_1 the situation changes. This is because c_1 does not only depend on d_1r_0, but also on d_0r_0 and d_0r_1. Hence the non-linearity increases and the correct d_1 is much more likely to have the highest correlation coefficient. Furthermore a wrongly guessed d_0 severely decrease the correlation coefficient in this step. Figure 6.2 shows the correlation coefficients for the hypotheses for d_1, where d_0 was chosen correctly. It can be seen that the right partial key $98d$ has the highest correlation. In Figure 6.3 the wrong positive was used for d_0. Hence the correlation coefficients are small. Finally Figure 6.4 shows, how likely it is that a wrong hypothesis is tolerated.

We overcame this problem by running the whole experiment for all 256 different values of d_0. At the end the signature generation algorithm can be run for every d. The correct signature will then indicate the correct d. One might assume that it is not necessary to rerun the experiment that often. However, in reality a wrongly guessed d_0 can not be detected so easily.
We ran the experiment with 1000 traces and extracted the key successfully. We have not

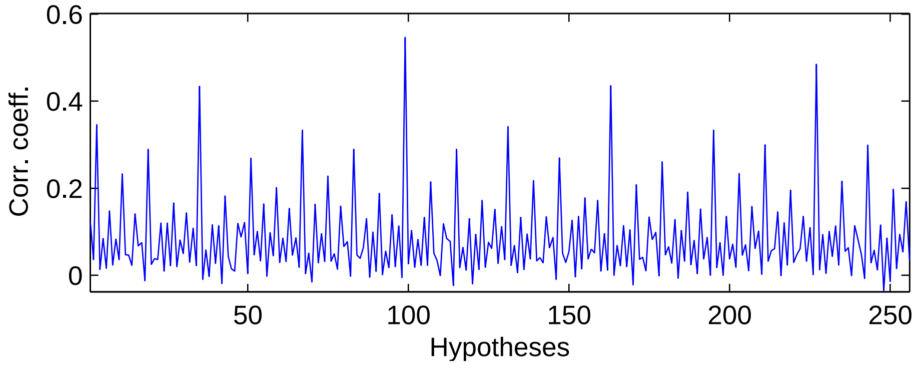


Figure 6.2: Correlation for the hypotheses for the second byte of d with a correctly guessed first byte.

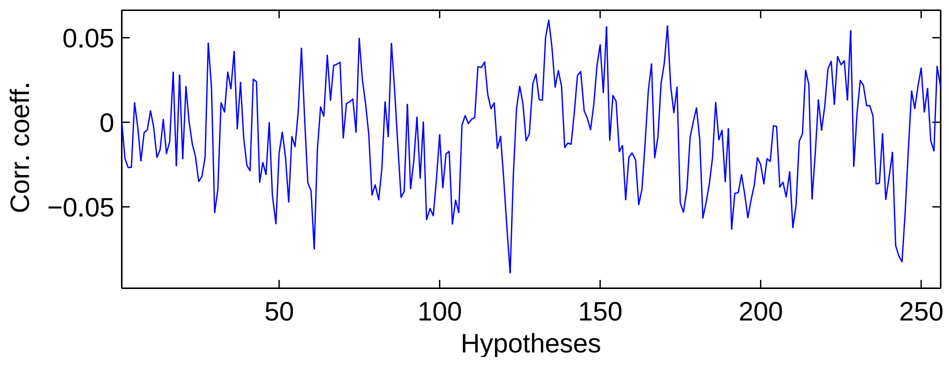


Figure 6.3: Correlation for the hypotheses for the second byte of d with a wrongly guessed first byte.

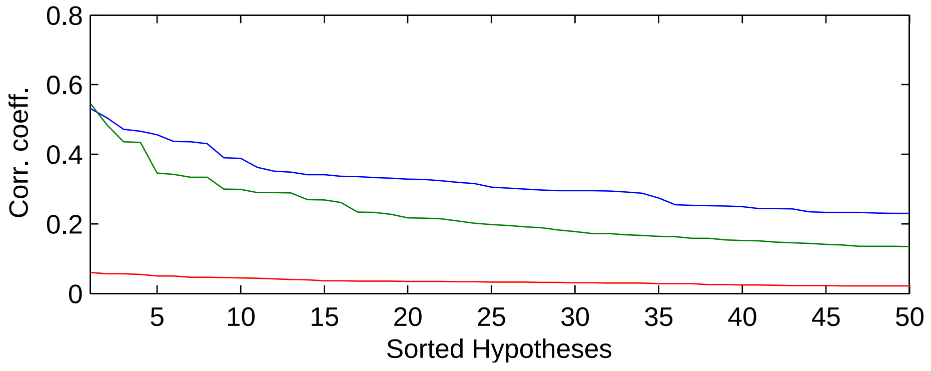


Figure 6.4: Sorted correlation coefficients for all three hypothesis. Blue corresponds to Figure 6.1, green to 6.2 and red to 6.3.

tried to find a lower bound for the number of needed traces, because only a single run needed several hours.

6.2 Template Attacks

In the previous section we needed a large number of traces to recover the private key. However, all attacks in this section have been successfully mounted with only one given trace from the target device. The target device is assumed to be a blackbox which takes a message and returns a signature. Besides that device, we are in possession of an identical device, for which we can choose the key as often as we want. This allows us to interleave the template building and matching steps. Assuming that we want to recover 10 bit, we therefore only need about 20 templates instead of 2^{10}. This is because we don't need to build them for all possible keys in advance. Instead we can just build two templates for every bit on the fly and then attack that bit. Even if we cannot interleave the building and the matching phase the attacks are feasible, since 10 bit are enough to recover k, given enough signatures.
For standard binary double and add algorithms it has been already shown in the past, that an ordinary SPA attack is feasible. Also in our setup an attack through visual investigation is possible. Furthermore we can confirm that SPA resistant algorithms make ordinary SPA attacks impossible. However, we show that this only holds for ordinary SPA attacks and is not true for template-based SPA attacks.
The targets of the attack differed in their EC group and underlying field arithmetic implementation. First we used normal binary double and add algorithms and mounted ordinary SPA attacks just to verify that our setup allows this kind of attack (Figure 5.2). Next we attacked double and always add as well as windowed algorithms. In the scenario in Section 6.2.1 the underlying field arithmetic had no countermeasures implemented and hence we could use key dependent timing differences as our source of leakage. For the implementation used in Section 6.2.2 we modified the field arithmetic in order to make the timing behavior key independent. However, it turned out that this leads to a DPA leakage which can again be used to build templates.

6.2.1 Ordinary Implementation

In this section we mainly take advantage of the fact that also SPA resistant algorithms still contain many data dependent branches. They occur in the point multiplication, the group operation and the field arithmetic algorithms. The most obvious ones are the reductions in the prime field arithmetic. The reductions with the greatest impact can be found in the prime field addition and prime field subtraction algorithms (Algorithms 3 and 4). Not only that they occur with a high probability, but also the algorithms themselves are used quite often (5 times during each doubling and 6 times during each addition). The impact of the final reduction during the fast reduction algorithm can be neglected. We used the fast reduction algorithm from [4] and hence the final reduction occurs only with a probability of about 2^{-32}.
The second information leakage we use for this attack is the presence of the different instruction classes of the used ARM processor. To show the significance of these two leakages, we acquired traces for two different scalars, i.e. two different ephemeral keys k, and calculated the difference.

In can be seen in Figure 6.5 that the voltage difference is about 400mV. Compared to the power consumption variance caused solely by the Hamming weight (Figure 2.6) this is a gain by a factor of 100. The only effect of the double and always add algorithm in Figure 6.6 is that the traces start to differ later, i.e. only one iteration after the first bit

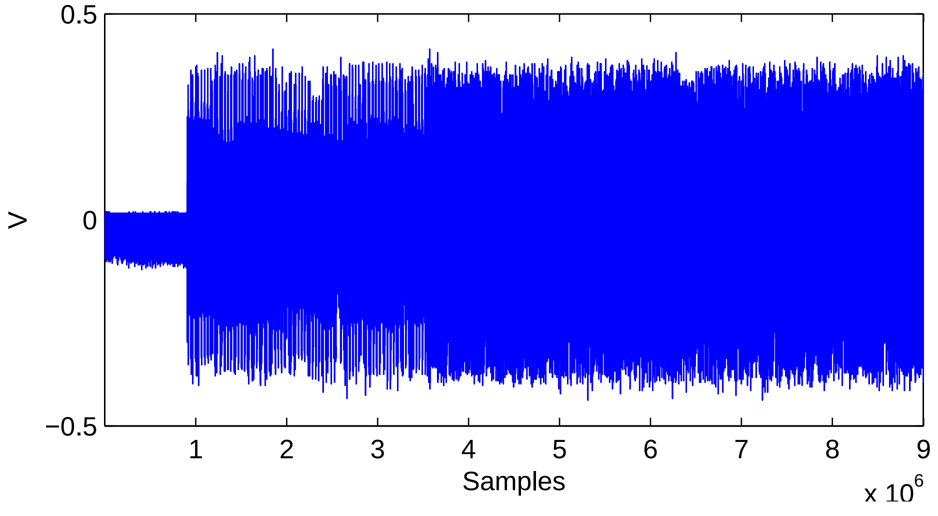


Figure 6.5: Difference trace for a double and add algorithm.

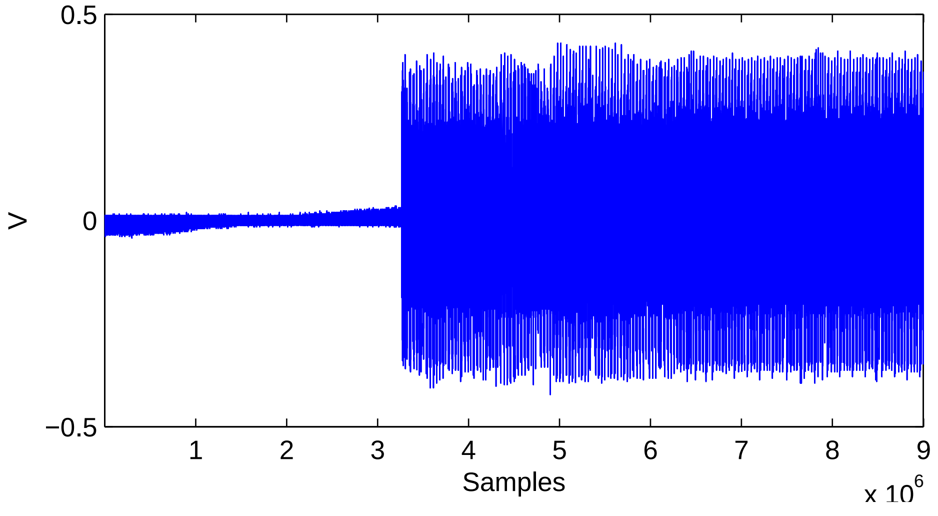


Figure 6.6: Difference trace for a double and always add algorithm.

of the scalar differs.
Equipped with this information we started to build the templates for the attack as following:

1. Build 2 templates which differ in the first interesting bit (the second most significant for Algorithm 14 and the least significant for Algorithm 15) of the scalar.

2. Calculate the difference of these traces and find the indices of the first n samples which differ most. The only thing which has to be considered is that we do not take samples of another bit into account.

3. These indices present our interesting points for the template. Hence we use those

Table 6.1: Matrix m containing the results of all pairwise template evaluations

	k_0	k_1	k_2	k_3
k_0	1	0	1	0
k_1	1	1	1	1
k_2	0	0	1	1
k_3	1	0	0	1

points for the characterization phase.

4. We test the given trace against these two templates and store the indicated key bit.
5. We acquire two traces for which the next bit differs and continue with step 2 until the whole key is recovered.

We used reduced templates and will also justify why. We are building templates for one specific key. Hence the intermediate values are always the same ones for one template, no matter how many traces we would acquire. If we neglect the electrical noise, then the power consumption must therefore always be the same. That means that there are no variances, which provide information. Hence if there are no variances for the single sample points, it follows that there cannot be any covariances between them.
Be aware that due to the high SNR in this attack scenario we just need two traces per template and nevertheless achieve a success rate of 1.

6.2.1.1 Extension for Window Methods

This attack can also be applied to windowed algorithms. For our attack we used a window size w of 4 bit. In this case we have to build $2^w = 16$ pairs of templates with 16 different traces. Furthermore the strategy has to be changed a bit. We cannot only choose one set of interesting points for all 16 templates and evaluate them altogether at once. In fact we could, but this would decrease the performance of our templates. Let $ip_{i,j}$ be the set of interesting points between the traces for the partial 4 bit keys pk_i, pk_j and the traces t_i, t_j respectively. If we would take $\bigcup ip_{i,j}$ into account for every template, we would characterize wrong features.
Therefor we build two templates for each $ip_{i,j}$ with t_i and t_j as the training set. Next we test the given trace against each pair and store the result in a matrix m containing the elements $m_{i,j}$. If the template for k_i yielded the higher probability we set $m_{i,j}$ to 1, otherwise a 0. If the trace was tested against 2 wrong templates then the outcome is random. However, if the correct template was involved, then this one should also yield the higher probability. At the end we get a matrix like in Table 6.1. In order to get the correct key, we just have to choose the row with the smallest variance.

6.2.1.2 Countermeasures

Due to the severe vulnerability of the implementation from Section 6.2.1, we implemented countermeasures which made the runtime constant.
The first approach was similar to the double and always add method: Every branch contained the same code and just the target variables differed. This basically made the overall runtime constant, but the moment, when the operations where executed still differed. If

we take a look at how a compiler translates *if*-*else* commands, it becomes clear why. First the condition is evaluated. If it evaluates to *true*, the processor executes the code in the *if* branch and then branches behind the code in the *else* branch. Otherwise it first branches and then executes the code.
The second approach was to insert NOPs at the beginning and at the end of each *if* and *else* branch. As a result the operations were executed at the same moment, independent of the key. However, if we take a look at two executions for different scalars, then the branch instructions overlap with NOPs. Since branch instructions are more power consuming than NOPs, there were still visible differences in the traces. They were only a few, but still they indicated data dependent branches.
The final and also successful approach was to trick around with pointer arithmetic. The basic idea can be seen in Algorithm 21.

Algorithm 21 Branches replaced by pointer arithmetic

```
int *arg_p $\leftarrow 0$
condition $\leftarrow$ (carry $= 1$)
arg_p $\leftarrow$ condition $\cdot$ modulus $+$ !condition $\cdot$ zero_vector
a $=$ long_sub(a, arg_p)
```

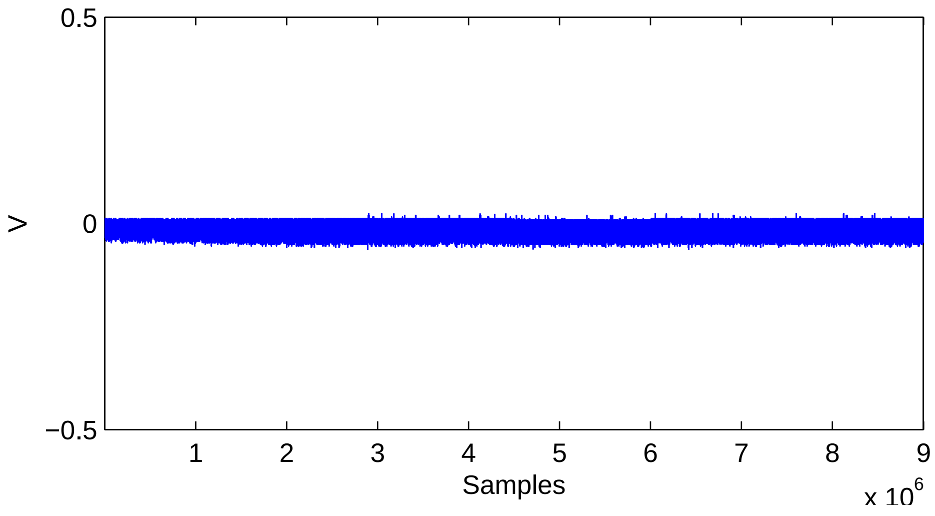


Figure 6.7: Difference trace for a constant runtime implementation.

Figure 6.7 shows that there are indeed no more peaks in the difference trace after the countermeasure has been implemented.

6.2.2 Constant Runtime Implementation

The countermeasure from before ensures that attacks which use timing differences are not possible anymore. However, if all instructions are always executed at the same moment in time, then also the intermediate values are processed at the same moment in time. This makes a DPA attack possible. That does not mean that we need many traces from the target device, but only from our identical device. We only want to find out when the

memory transfers occur relatively to the start of the scalar multiplication. The start itself can be found with the help of pattern matching. Since a scalar multiplication always starts with the same sequence of instructions we can define such a pattern. Then we do a cross correlation between the given trace and the pattern.
In the case of the P192 curve, each EC group operation yields about 100 intermediate values. For curves over larger fields we get even more. Further we can predict these occurring intermediate values by gradually recovering the key, as done before. Hence we can build templates for them and detect their occurrence. A correlation trace for an EC group operation and the occurring intermediate values can be seen in Figure 6.8.

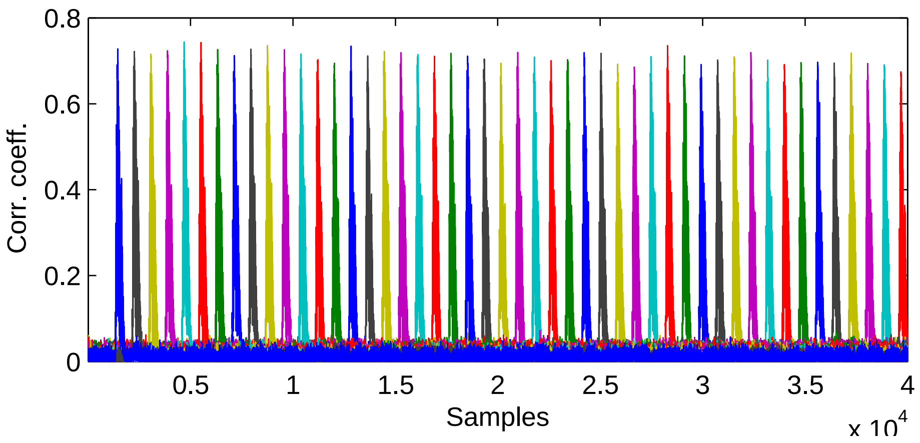


Figure 6.8: Correlation trace for the processed intermediates during a group operation

6.2.2.1 Template building

The fact that we have so many intermediate values to attack is very helpful for two reasons. First it makes the point selection much easier. We do not need heuristics in this case. Instead we can just use the highest DPA peak for each intermediate as our points of interest. Second, we do not need to use all of the intermediate values. Instead we only use those for which the Hamming distance between the two keys is large. Therefor we need some heuristics, but this parameter is rather easy to determine. If the Hamming distance is too small, the SNR becomes too small. On the other hand, if it is too large, the number of intermediates which fulfill this requirement becomes too small. However, due to the distribution of the Hamming distance, this number decreases rapidly from a certain point. In our case four was a suitable choice.
After having found suitable intermediates to attack, we can start to build the templates. We use reduced templates again. There are two arguments for this approach. First, we attack the Hamming weights of intermediate values, which are part of a long integer vector. This leads to the assumption that these values are rather independent of each other. In order to confirm this assumption we made some experiments: We generated 1000 random 192 bit operands a and b and used them for various calculations. Table 6.2 shows the maximal occurring covariances. In the first row only the covariance between the words of the operands was considered. Since they were chosen randomly, such a small value had to be expected. However, the maximum for the covariances between the operands

Table 6.2: Maximal covariance between the 32-bit words for different calculations

$cov(a, b)$	0.5544
add	0.5969
add mod p	0.6332
mul	0.9941
mul mod p	0.6234

and the result is not significantly larger. The slightly larger value for the multiplication occurs between the most significant words of the operands and the result. This is quite reasonable since a normal integer multiplication shows a little correlation as well. They second argument is the same as before: The processed values are always the same ones.

Furthermore we only need to build one set of templates for the memory transfer instruction and can reuse it for every intermediate value. We can either build one big template by duplicating the set we already have or we can use one template for every intermediate value and combine the results lateron. According to Equation 5.1 this makes no difference.

6.2.2.2 Template evaluation

When we evaluate the templates we can make the following observation. A single template, i.e. a template for only one intermediate, has a performance of only about 0.65. Hence the probability that we guess just a single key bit correctly is rather small. However, if we think in terms of 192 bit keys the success probability to recover the key is below 10^{-30}. Even if we just want to recover 10 bits for a lattice based attack, the probability is only 0.013. Not to mention that we need more than one key chunk for a lattice attack.
However, if we attack not only one intermediate value, but several at once, the success rate approaches one. In fact, if the algorithm provides enough intermediates, a single template only needs a performance greater 0.5. For our case it can be seen in Figure 6.9 that we need about 40 intermediate values to achieve a success rate which lies way beyond 0.99. That means, it is theoretically high enough to recover the whole key. The only problem with that is the insufficient memory of the scope we used. However, there are already scopes by Tektronix for instance which are able to acquire 400MS and hence a complete scalar multiplication.

6.2.2.3 Extension for Window Methods

The extension of this attack for window based algorithms can be done analogous to before. The interesting points for every pair of partial keys differ as well, since we choose different suitable intermediate values.

6.3 Countermeasures

We have shown that our attacks overcome SPA resistant implementations as well as constant runtime implementations. Furthermore there are even some blinding techniques which do not prevent our attacks. For instance scalar blinding and point blinding. In the case of scalar blinding one adds the order of the base point multiplied by a random number to the original scalar. Such a key $k' = k + rand * ord(P)$ is congruent k. Hence k' still

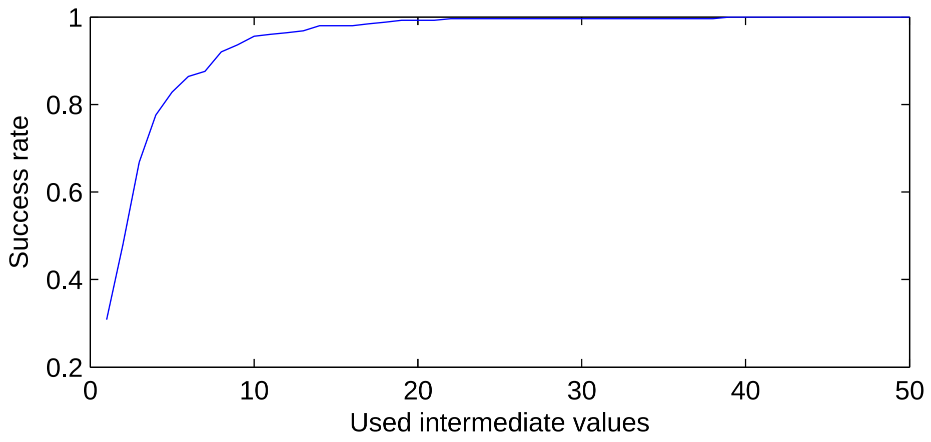


Figure 6.9: Template success rate depending on the number of used intermediate values. The success rate for one intermediate is only 0.3, because not all traces in the test set had a sufficient high Hamming distance for the first intermediate. Hence they were not tested and considered as unsuccessful.

satisfies the ECDSA signing equation and therefore can be used to recover d. However, due to the larger k' the complexity for lattice attacks would grow. This does not make the attack infeasible. It just requires larger chunks of k' and therefore maybe a more powerful acquisition equipment. In the case of base point blinding one uses a $P' = rand * P$ and calculates the signature with $s = (k * rand)^{-1}(e + dr) \mod n$. With our attack applied twice we can also overcome this countermeasure by attacking both scalar multiplication. However, the scalar $rand$ has to be recovered completely for every signature. Recall that our attack is capable of that if the acquisition equipment has a sufficient large memory.
Countermeasures which prevent our attack are those which prevent DPA attacks. One such example is the base point randomization. If the used algorithm for the EC point addition allows the use of projective representatives for both points, the base point can be randomly projected. Hence we cannot predict the occurring intermediate values and our attack fails.

Chapter 7

Conclusions

In this thesis we combined several (new) methods in order to develop a powerful template-based SPA attack against ECDSA implementations. The underlying platform was a common 32-bit ARM based microcontroller as often used in mobile devices. The attack not only breaks implementations secure against other types of SPA attacks, but can also overcome some other countermeasures.
The attack elaborated in this thesis is interesting and practical for several reasons. First, the algorithm is used in practice and therefor presents a realistic target. Second, the base point is fixed, hence we can guess the occurring intermediates and restrict the number of needed templates to a minimum. Third, ECDSA is vulnerable to lattice-based attacks, therefore it is sufficient to only recover some of the key bits per signature. This further enables the attack even if the acquisition of the whole EC point multiplication is not possible.
For the practical implementation of the attack we built templates according to the processor's power model. We further applied the templates to sequences of intermediate EC points. Finally we elaborated criteria in order to restrict the choice of used intermediate points and further increase the performance of the attack. In the end it was shown that we only need one trace to recover the private key with a success probability close to one. Due to the nature of the attack, the only countermeasures preventing the attack are those, which randomize the intermediate curve points and hence prevent DPA.
To the best of our knowledge this thesis presented the first practical and successful results of template-based SPA attacks on asymmetric cryptographic systems.

Bibliography

[1] E. T. Adi Shamir. Acoustic cryptanalysis. on nosy people and noisy machines. html.

[2] C. Archambeau, E. Peeters, F.-X. Standaert, and J.-J. Quisquater. Template attacks in principal subspaces. In *CHES*, pages 1–14, 2006.

[3] J.-S. Coron and L. Goubin. On Boolean and arithmetic masking against differential power analysis. *Lecture Notes in Computer Science*, 1965:231–??, 2001.

[4] J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 133–147. Springer Verlag, 2004.

[5] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] N. A. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.

[7] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[8] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996.

[9] T. Le, J. Clediere, C. Serviere, and J.-L. Lacoume. High order statistics for side channel analysis enhancement. 2006.

[10] S. Mangard. Exploiting Radiated Emissions EM Attacks on Cryptographic ICs. In T. Ostermann and C. Lackner, editors, *Austrochip 2003, Linz, Austria, October 1st, 2003, Proceedings*, pages 13–16, 2003. ISBN 3-200-00021-X.

[11] P. T. Mangard Stefan, Oswald Elisabeth. *Power Analysis Attacks.* Springer, 2007.

[12] T. S. Messerges. Using second-order power analysis to attack dpa resistant software. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 238–251, London, UK, 2000. Springer-Verlag.

[13] T. S. Messerges. Securing the aes finalists against power analysis attacks. In *FSE '00: Proceedings of the 7th International Workshop on Fast Software Encryption*, pages 150–164, London, UK, 2001. Springer-Verlag.

[14] E. Oswald and M. Aigner. Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2001.

[15] E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[16] Philips Semiconductors. *LPC2114/2124/2212/2214 USER MANUAL*, May 2003.

[17] T. Popp and S. Mangard. Masked Dual-Rail Pre-Charge Logic: DPA-Resistance without Routing Constraints. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, Scotland, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2005.

[18] C. Rechberger and E. Oswald. Practical Template Attacks. In C. H. Lim and M. Yung, editors, *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23 – 25, 2004, Revised Papers*, volume 3325 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2004.

[19] W. Schindler, F. Koeune, and J. Quisquater. Unleashing the full power of timing attack, 2001.